NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPS AUV INTEGRATED SIMULATOR

by

Donald P. Brutzman

March, 1992

Thesis Advisors: Yutaka Kanayama Michael J. Zyda

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
NPS AUV INTEGRATED SIMULATOR

12. PERSONAL AUTHOR(S)
Donald P. Brutzman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 04/90 TO 03/92	1992, March, 17	269

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Autonomous Underwater Vehicles, Graphics, Simulation, Path Planning, Sonar Classification, Expert Systems, Real-Time Operating Systems.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The development and testing of Autonomous Underwater Vehicle (AUV) hardware and software is greatly complicated by :hicle inaccessibility during operation. Integrated simulation remotely links vehicle components and support equipment with aphics simulation workstations, allowing complete real-time, pre-mission, pseudo-mission and post-mission visualization in e lab environment. Integrated simulator testing of software and hardware is a broad and versatile method that supports rapid d robust diagnosis and correction of system faults. This method is demonstrated using the NPS AUV.

High-resolution three-dimensional graphics workstations can provide real-time representations of vehicle dynamics, control stem behavior, mission execution, sensor processing and object classification. Integrated simulation is also useful for velopment of the variety of sophisticated artificial intelligence applications needed by an AUV. Examples include nar classification using an expert system and path planning using a circle world model.

The flexibility and versatility provided by this approach enables visualization and analysis of all aspects of AUV velopment. Integrated simulator networking is recommended as a fundamental requirement for AUV research and velopment.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: r. Yutaka Kanayama and Dr. Michael J. Zyda	22b. TELEPHONE (Include Area Code): (408) 646-2095/2305; 22c. OFFICE SYMBOL: CS/37

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED

Approved for public release; distribution is unlimited.

NPS AUV INTEGRATED SIMULATOR

by

Donald P. Brutzman
Lieutenant Commander, United States Navy
B.S.E.E., United States Naval Academy, 1978

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

The development and testing of Autonomous Underwater Vehicle (AUV) hardware and software is greatly complicated by vehicle inaccessibility during operation. Integrated simulation remotely links vehicle components and support equipment with graphics simulation workstations, allowing complete real-time, pre-mission, pseudo-mission and post-mission visualization and analysis in the lab environment. Integrated simulator testing of software and hardware is a broad and versatile method that supports rapid and robust diagnosis and correction of system faults. This method is demonstrated using the Naval Postgraduate School (NPS) AUV.

High-resolution three-dimensional graphics workstations can provide real-time representations of vehicle dynamics, control system behavior, mission execution, sensor processing and object classification. Integrated simulation is also useful for development of the variety of sophisticated artificial intelligence applications needed by an AUV. Examples include sonar classification using an expert system and path planning using a circle world model.

The flexibility and versatility provided by this approach enables visualization and analysis of all aspects of AUV development. Integrated simulator networking is recommended as a fundamental requirement for AUV research and deployment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Many people unselfishly contributed to the work contained in this thesis and I wish to gratefully acknowledge their assistance.

LCDR Mark A. Compton USN coauthored two chapters, two papers and the expert system source code. His expert knowledge and sound judgement were always available to discuss any aspect of integrated simulation. In one short week he adapted his minefield search strategy to utilize integrated simulation data passing mechanisms, proving the accessibility of this method for graphics simulation display. I am grateful for our opportunity to work together.

CDR Charles A. Floyd USN assisted in the implementation of parametric regression and helped me learn many of the nuances of our vehicle and simulation programs. CDR Thomas A. Jurewicz USN blazed the trail with his dynamic simulator. LT R. Scott Starsman USN derived the final form of Equation (7.1). Charles Lombardo of the NPS technical staff provided frequent good advice on programming in C and Unix. Russell Whalen dependably provided underwater photography, pool test support, boundless knowledge about how to make things work and enthusiastic encouragement. He and Walt Landaker built the laboratory AUV, networked it and made it listen to our commands.

I thank my former Commanding Officer CAPT Alan R. Beam USN of DARPA and Mr. Patrick Hale of Charles S. Draper Laboratories for access to the DARPA UUV and the UUV support simulator.

Interaction with the many professors and students in the NPS AUV research group is always intellectually stimulating. Dr. Se Hung Kwak, Dr. Neil C. Rowe, Dr. Michael L. Nelson MAJ USAF, Dr. Fotis Papoulias, Dr. Shridhar Shukla and Dr. James Clynch provided valuable critical analysis. Dr. Richard W. Hamming has provided many stimulating ideas on the importance of working on projects of value.

NPS AUV project leaders Dr. Anthony M. Healey and Dr. Robert B. McGhee are particularly thanked for their continuing guidance and inspiration.

My thesis advisors are two of the most impressive people that I have ever met. Dr. Yutaka Kanayama's knowledge of spatial reasoning and robotics is unparalleled. His ability to discern the heart of a problem and uncover fundamental principles has been inspiring, and his patience and accessibility has been invaluable.
Dr. Michael J. Zyda provided the knowledge and tools to make real-time visual simulation a reality. He is always looking three steps ahead at where we ought to go next. I am indebted to both men.

The biggest ingredient spent in this thesis has been time: time to explore new ideas, time to understand problems, time to wrestle uncooperative software into shape, time to write results in an understandable way. My time was spent away from my loving wife Terri and my three wonderful daughters Hilary, Rebecca and Sarah.
I dedicate this work to them as a small "thank you" for their support.

I. INTRODUCTION

A. PROBLEM STATEMENT

Designing, building and testing an Autonomous Underwater Vehicle (AUV) is difficult. AUVs must operate unattended and uncontrolled in a remote and unforgiving environment. Inaccessibility greatly complicates evaluation, diagnosis and correction of AUV system faults. In order to ensure complete reliability, AUV software and hardware need to be fully tested in the laboratory before operational deployment. Such important testing requirements cannot be met using only a standalone AUV.

Designing an AUV is also complex. Many capabilities are required for a mobile robot to act independently. Sensing, motion control, motion planning, mission planning, failure recovery and overall control are all essential. Interaction between vehicle processes and the mechanics of actual implementation must also be solved. These complex problems cannot be modeled, simulated or integrated into an autonomous mobile robot without understanding their fundamental principles and difficulties.

The primary problem addressed by this thesis is how to design and construct an integrated simulator in order to completely visualize AUV performance in support of distributed research and technical evaluation. All aspects of AUV software design and simulation are considered. As direct examples of how integrated simulation may be applied, in-depth analysis is also provided for the future roles of naval AUVs, sensor analysis, path planning and real-time interaction.

B. MOTIVATION

The principal motivation driving the development of an AUV integrated simulator is to meet the research needs of the large academic group working on the Naval Postgraduate School (NPS) AUV. Students and professors alike have diverse research goals that are often forced to compete for access to vehicle system software and limited pool test time. The need to use operational software running on actual NPS AUV hardware is a particularly important requirement. Lack of accessibility to the NPS AUV in a distributed laboratory environment has occasionally prevented implementing new software applications on the vehicle. Pre-mission validation of vehicle systems response to new software has been similarly limited in scope, resulting in several operational test failures and frustrating delays in development. Integrated simulation is a high-level tool that enables solutions to all of these challenges.

Scientific visualization of complex interactions greatly improves our understanding of how things work. Human beings are visually oriented. Being able to see and control a moving picture allows us to quickly and intuitively understand numerous process interactions. A fundamental computer science tenet first expressed by Dr. Richard A. Hamming is that "The purpose of computing is insight, not numbers" (Hamming 73). Integrated simulation is intended to provide insight.

The term artificial intelligence typically refers to the study of how to perform tasks that are usually considered to require human intelligence. Numerous such artificial intelligence tasks are required for a mobile robot to achieve autonomy. The importance of solving artificial intelligence problems is widely recognized. Prominent robotics researcher Hans P. Moravec states,

> "...solving the day to day problems of developing a mobile organism steers one in the direction of general intelligence, while working on the problems of a fixed entity is more likely to result in very specialized solutions... Mobile robotics may or may not be the fastest way to arrive at general human competence in machines, but I believe it is one of the surest roads." (Moravec 83)

This study of integrated simulation has helped reveal valuable conclusions regarding the artificial intelligence subjects of sensor analysis and path planning.

The NPS AUV Integrated Simulator has been designed to support complete scientific visualization of actual NPS AUV vehicle performance. The lessons learned while building this integrated simulator have proven that distributed research can be effectively accomplished when proper network connections and data-passing mechanisms are provided. The integrated simulation approach has great value and general applicability for the rapid development of all types of mobile robots.

C. OBJECTIVES

This thesis addresses the following research questions:

- How can an integrated simulator be constructed to support pre-mission, pseudo-mission and post-mission AUV evaluation?
- How can integrated simulation support distributed research?
- What is required to allow both local and remote mission performance analysis?
- How can a laboratory AUV microprocessor be used for preliminary AUV testing in a distributed research environment?
- How can the numerous processes that make up NPS AUV control software communicate with the NPS AUV graphics simulation program?
- How can inter-process communication be accomplished identically and independently regardless of where AUV software is running?
- How can the Computer Science Department Network, Campus-Wide Network and NPS AUV be linked together to connect the variety of processors and operating systems that support the NPS AUV?
- How can active sonar range and bearing data be analyzed to classify sonar contacts?
- How can a shortest path be found around circular or cylindrical obstacles?
- What real-time operating system considerations must be met in order to support parallel operation by mutually-cooperating artificial intelligence AUV applications interacting with a real-time environment or a near-real-time simulation?

D. THESIS ORGANIZATION

Many components and many concepts make up an integrated simulator. Conduct of this research led down many interesting intellectual trails. Consequently the scope of material contained in this thesis is broad while the many individual conclusions are detailed. Given this diversity of material, the objectives, summaries of previous work and conclusions are included with each pertinent chapter. When appropriate, thesis chapters have doubled as separate articles to report on results of general interest. The five sections that have also been written for independent publication are identified below, as are the contributing coauthors.

Chapter II describes AUV research at NPS. It was originally written as a survey article to summarize the many facets of AUV research at NPS, as well as describe expected future roles of naval AUVs. This chapter is an important prelude to the thesis in that it establishes the scope of current and future AUV work that an integrated simulator must support. This chapter appeared as an article in *Sea Technology* and was cowritten with LCDR Mark A. Compton USN (Brutzman Compton 91).

Chapter III defines and develops the concept of rapid simulation for rapid development of AUVs. As such it is the heart of this thesis. This chapter is accepted for presentation to the IEEE Oceanic Engineering Society Symposium on Autonomous Underwater Vehicles 1992. Dr. Yutaka Kanayama and Dr. Michael J. Zyda are coauthors of the corresponding article (Brutzman Kanayama Zyda 92).

Chapter IV presents design specifications that are pertinent to the specific integrated simulator implemented for the NPS AUV. Chapter V describes specific data network requirements for the NPS AUV Integrated Simulator. Details are included that are based on the difficulties and successes encountered during implementation. The technical skills and determined efforts of Mr. Russell Whalen and Mr. Walter Landaker were instrumental in building the laboratory AUV and connecting it to the network.

Chapter VI presents fundamental work on the effective synthesis of geometric analysis and expert system heuristics for classifying underwater objects. This chapter was also submitted in article form for presentation to the IEEE Oceanic Engineering Society Conference *OCEANS 92*. This work was cowritten with LCDR Mark A. Compton USN and Dr. Yutaka Kanayama (Brutzman Compton Kanayama 92). Examples of applicability to integrated simulation are included.

Chapter VII presents fundamental work on shortest path planning using circular obstacles. This work is based primarily on Dr. Yutaka Kanayama's theories of optimal robot motion. Examples of applicability to integrated simulation are included in the chapter.

Chapter VIII describe real-time operating system considerations, all of which are pertinent both to an operating AUV and an AUV integrated simulator. This chapter has also been included in a NPS technical report (Badr Byrnes Brutzman Nelson 92) and was edited by Dr. Michael L. Nelson, MAJ USAF.

Chapter IX presents NPS AUV Integrated Simulator limitations and performance measurements. Chapter X discusses the many promising opportunities for potential future research and follow-on work using the NPS AUV Integrated Simulator. Chapter XI presents conclusions and recommendations.

Appendix A is a guide for NPS AUV users and software developers who are interested in utilizing the NPS AUV Integrated Simulator. Appendix B is a synopsis of the lengthy source code written for the graphics simulation component of the NPS AUV Integrated Simulator.

Appendix C is the "C" Language Integrated Production System (CLIPS) expert system source code for NPS AUV Sonar Classification System. This program demonstrates the concepts described in Chapter VI and was cowritten with LCDR Mark A. Compton USN.

Appendix D reproduces an as-yet-unpublished paper cowritten with Dr. Yutaka Kanayama that provides mathematically rigorous theoretical detail on circle world

shortest path planning (Kanayama Brutzman 91). Appendix E provides source code implementation of the circle world path planning algorithms.

Appendix F describes how NPS AUV Integrated Simulator source code may be obtained via Internet. Graphics simulation, sonar classification and path planning software programs are all available.

Appendix G is a videotape demonstration of pertinent thesis results. Video is essential to portray the power and effectiveness provided by real-time graphics simulation and scientific visualization techniques. This videotape appendix includes a short segment on the NPS AUV presented at the IEEE Robotics and Automation Conference 1992 (Brutzman Floyd Whalen 92). The short segment was coproduced with CDR Charles A. Floyd USN and Mr. Russell Whalen, and benefited from the technical advice of Dr. Michael J. Zyda and Mr. David Pratt.

II. AUV RESEARCH AT THE NAVAL POSTGRADUATE SCHOOL

Naval officers and civilian scientists at NPS are working on an AUV that is helping change the nature of undersea warfare. Under the guidance of faculty in the Computer Science, Mechanical Engineering and Electrical Engineering departments, NPS students have developed and built a working AUV that can maneuver and operate submerged and unattended. The NPS AUV is a robotic platform for basic research and thesis work in control technology, artificial intelligence, computer visualization, and systems integration.

A. IMPORTANCE OF AUVs IN NAVAL MISSIONS

Unmanned autonomous submersibles have many characteristics that make them particularly attractive for employment in naval missions (Polmar 91). Vehicle autonomy allows independent operation in changing situations without a tether or any direct human intervention. Removing the need for a crew permits routine operation in extremely deep, shallow, or tactically hazardous environments, and also eliminates the requirements for large and expensive support equipment. Small size and quiet propulsion systems result in unmatched stealth. The relatively low cost of AUVs enables the acquisition of many units that might serve as force multipliers for each of the Navy's warfare communities.

Many robotic vehicles are already deployed in the fleet and saw action in Operation Desert Storm. Tomahawk cruise missiles are autonomous weapons that inflicted heavy damage with precision accuracy (Arthur 91). Remotely piloted vehicles (RPVs) flew with great success in reconnaissance and naval gunfire support missions, to the extent that Iraqi soldiers initially made RPVs priority targets but later surrendered to them (Arthur 91) (Burke 91). Remotely operated vehicles (ROVs) controlled by minesweepers have the capability to send back live underwater video in order to aid in the hazardous and time-consuming job of classifying and deactivating

mines (Polmar 87). Adding autonomy to unmanned vehicles dramatically increases their independent operating range and tactical capabilities. The type of research work being conducted at NPS is of fundamental importance in making these performance breakthroughs possible for unmanned underwater vehicles.

B. NPS AUV DESIGN SPECIFICATION SUMMARY

The design and construction of the first NPS AUV began in 1987. NPS AUV I was a two-foot prototype model with operational screws and gyros that was used for the investigation of model-based maneuvering controls, including the automatic identification of significant hydrodynamic characteristics (Healey 89). The full-scale vehicle NPS AUV II was constructed by the students, technical staff and faculty of the Mechanical Engineering department, and required over a year to design and build (Good 89). This AUV was launched (complete with traditional champagne christening!) by the Superintendent, RADM Ralph W. West, Jr. at the NPS pool on June 15, 1990. According to Admiral West, a submarine officer,

> "The AUV is one of the new technologies that will play a major role in maintaining the effectiveness of our fleet units as the threats facing us become more sophisticated and diverse." (West 91)

The current NPS AUV is eight feet long and neutrally buoyant, displacing 387 pounds (Figure 2.1). Its overall size and shape is comparable to a dolphin. Current vehicle endurance is two to three hours. Maximum speed of the NPS AUV is about two knots. The NPS AUV's turning diameter is under three body lengths, designed to be ideal for maneuvering in the large NPS swimming pool (Figure 2.2). The low noise level in the NPS pool allows precise testing in a controlled environment.

Specific test missions are downloaded into the NPS AUV using a poolside laptop computer (Figure 2.3). NPS AUV posture and sonar data are recorded ten times per second throughout each mission and then immediately uploaded afterwards for post-mission analysis. By following mission software commands, the NPS AUV can change course and depth without any external direction. The low speed of the vehicle allows divers to swim nearby and safely evaluate its performance (Figure 2.4).

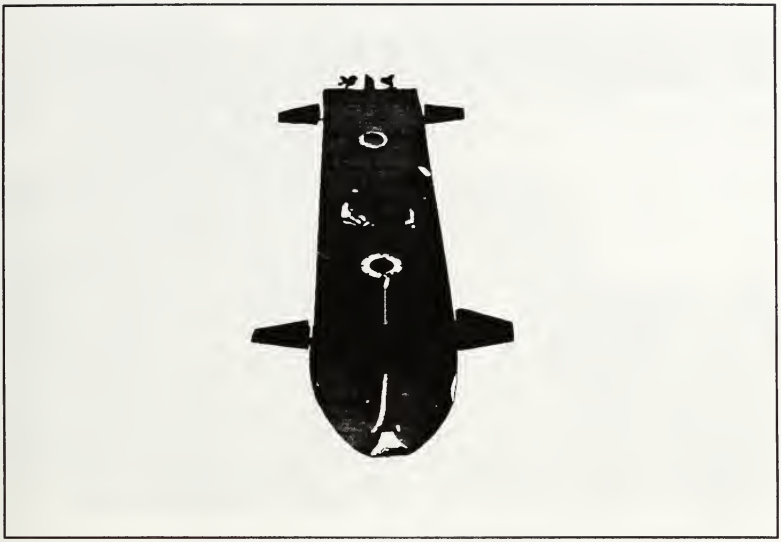


Figure 2.1 The NPS AUV is an eight foot long submersible

Open-ocean testing is feasible but will be reserved for a more robust follow-on vehicle.

Initial AUV project objectives include the study of mission planning, navigation, collision avoidance, real-time mission control and replanning, object recognition, vehicle dynamic response and motion control, and post-mission data analysis. The primary components of the AUV are an aluminum hull, fiberglass sonar dome, four high-frequency directional sonar transducers, twin counter-rotating four-inch propellers, lead-acid batteries, eight plane surfaces, and a Gespac computer running a Motorola 68030 processor with a 2MB RAM card. Four cross-body tubes have been included to house a new type of thruster that is under development. When completed, these thrusters will allow the AUV to control vehicle posture and maintain station in the presence of underwater currents. Figure 2.5 shows a general schematic of the AUV.

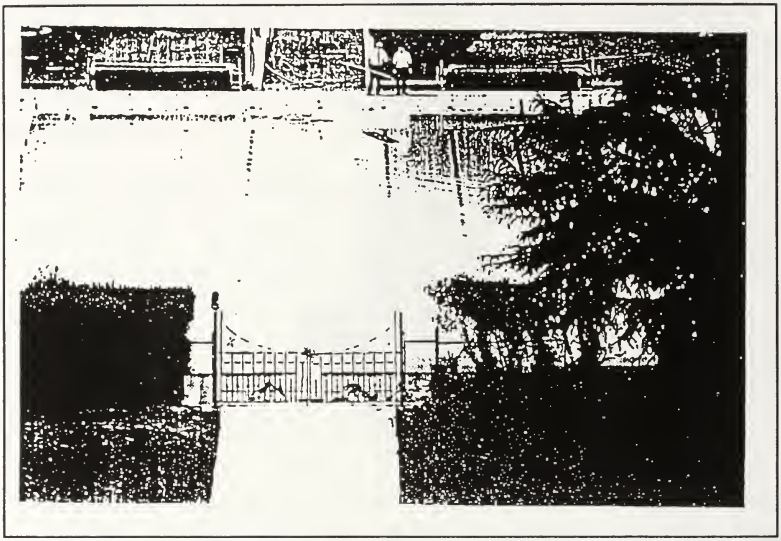


Figure 2.2 The NPS swimming pool is an ideal test environment for the NPS AUV

C. NPS AUV RESEARCH OBJECTIVES

The scope and missions of the AUV project are not restricted by any specific programmatic requirements. Dr. Robert B. McGhee, Computer Science department chairman states,

> "The primary purpose of our AUV work has always been to support student thesis and dissertation research without restrictions." (McGhee 91)

Given this climate of academic freedom, a myriad of topics are under active investigation and are resulting in numerous advances in underwater vehicle technology. Already over fifty theses and research papers have been published about the NPS AUV.

The Mechanical Engineering department has primary responsibility for the design, construction and operation of this submersible robot. Dr. Anthony J. Healey is the Mechanical Engineering department chairman and the AUV project principal

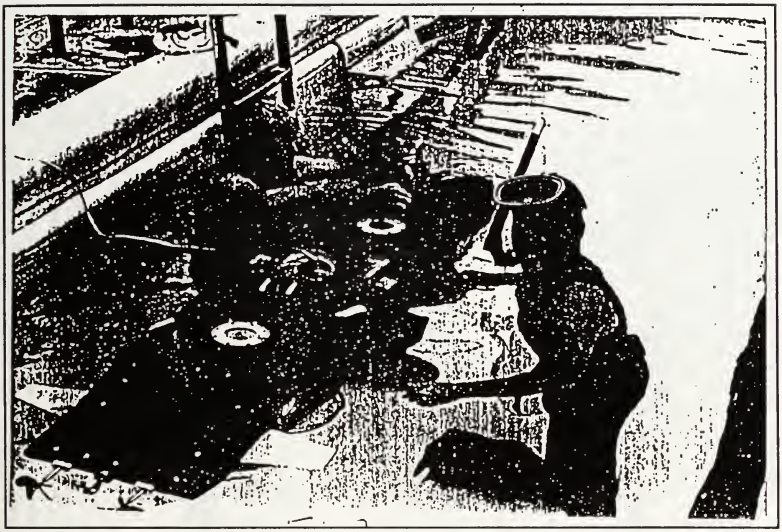


Figure 2.3 Specific test missions are downloaded into the NPS AUV using a poolside laptop computer

investigator. He has assembled a diverse group of over two dozen faculty and students, making this project the largest group research effort at NPS. Dr. Healey states,

> "The most important aspect of the project is to involve naval officer students in the development of control technology for future AUVs, utilizing their considerable experience. It also educates them in the potential capabilities of AUVs and the technical difficulties yet to be solved." (Healey 91)

Mechanical Engineering and Electrical Engineering department research is currently investigating vehicle stability and control, modeling submerged dynamic behavior, systems integration and guidance/autopilot design. Of immediate interest is integrating low-power components such as ultrasonic sonars, steering and diving controllers, guidance circuitry, Global Positioning System (GPS) receivers, and miniaturized inertial measurement units currently used for cruise missile navigation.

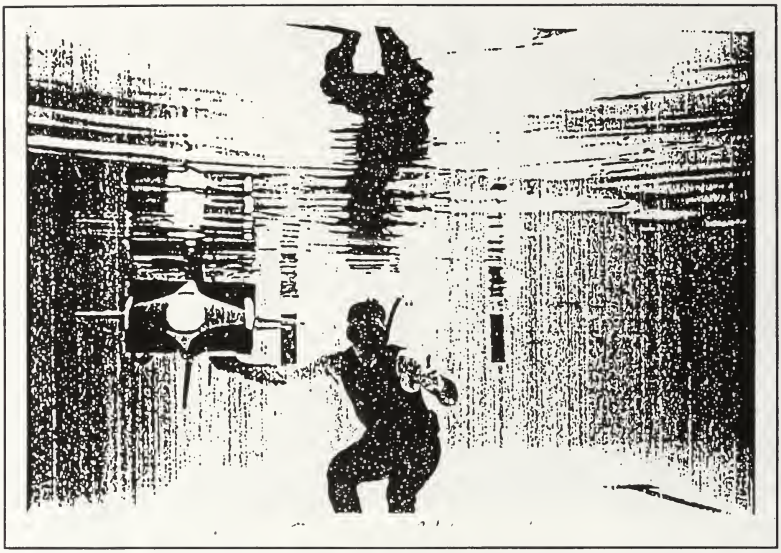


Figure 2.4 The low speed of the NPS AUV allows divers to swim nearby and evaluate its performance

Computer science programmers are designing the "brains" of the AUV. This robot must be able to transit independently to the desired operating area, perform a mission, return and report despite any unpredictable tactical situations that might occur. Real-time mission planning and obstacle avoidance are critical aspects of these tasks. The basic operations of the AUV resemble those found in naval ships and aircraft. However, piloting and tactical functions normally performed by humans must be independently handled by the AUV's on-board computer. Not surprisingly, AUV software organization is similar to a ship's underway watch team. Navigation, obstacle avoidance, data collection and mission execution must all occur continuously and in real time (Healey et al. 91). Figure 2.6 is a block diagram of the AUV mission execution software structure. Most of the functions represent tasks normally performed by human operators on submarines.

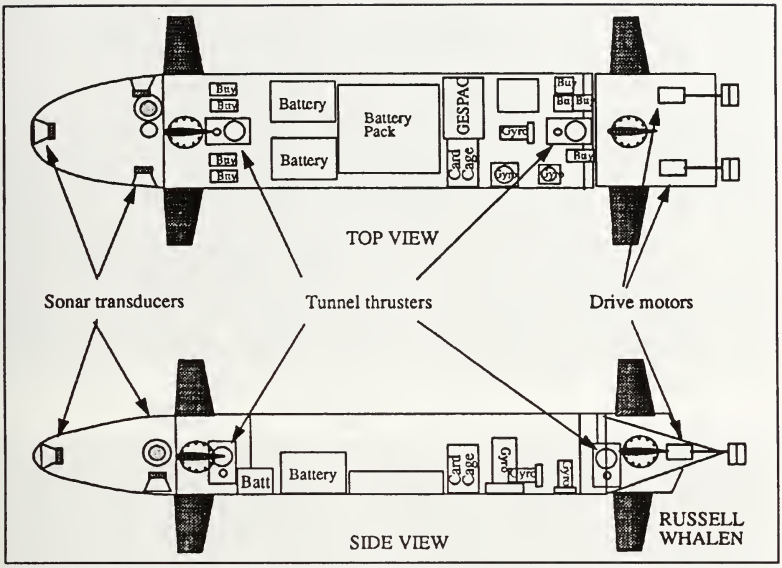


Figure 2.5 General schematic of the NPS AUV. Note the twin screws, four sonar transducers forward, and eight plane surfaces

Real-time three-dimensional computer graphics simulation of the AUV is being used for extensive laboratory evaluation. New hardware and software can be tested prior to installation and operation, minimizing risk and saving time and money. Replays of actual data recorded by the AUV can be used for visualization of remote environments and detailed post-mission data analysis. Silicon Graphics Inc. workstations identical to those used for special effects in movies such as *Terminator 2: Judgement Day* (Myers 91) are networked together and provide massive processing power. Because the AUV hull shape is similar to the Swimmer Delivery Vehicle used by Navy special warfare SEAL teams, a sophisticated mathematical model was already available for simulator use to accurately recreate vehicle dynamic motion and response characteristics (Zyda 90). This dynamics model has been validated by pool testing. NPS AUV graphics simulations can use actual hydrographic

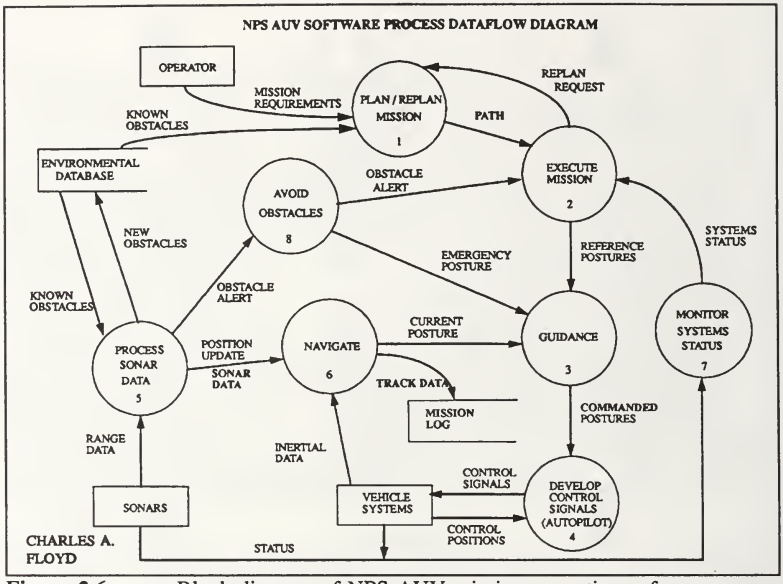


Figure 2.6 Block diagram of NPS AUV mission execution software structure

sounding data from Monterey Bay provided by the U.S. Geological Survey to show the detailed level of display and analysis possible (Figure 2.7) (Jurewicz 91).

Display of sonar beams and the sonar environment can also assist operators in evaluating AUV performance (Figure 2.8). Multiple simulation features can be combined in an integrated simulator. Complete visualization of the ocean environment and AUV system response permits sonar data post-mission analysis, precise hydrodynamics modeling and extensive software testing to be performed in real time.

Artificial intelligence techniques allow a robot to perform tasks normally requiring human intelligence. Many artificial intelligence methods are being developed for the AUV. Path planning and spatial reasoning are used to determine how to avoid obstacles and optimally travel from one location to another. Mission planning enables the AUV to execute an ordered mission, while mission replanning

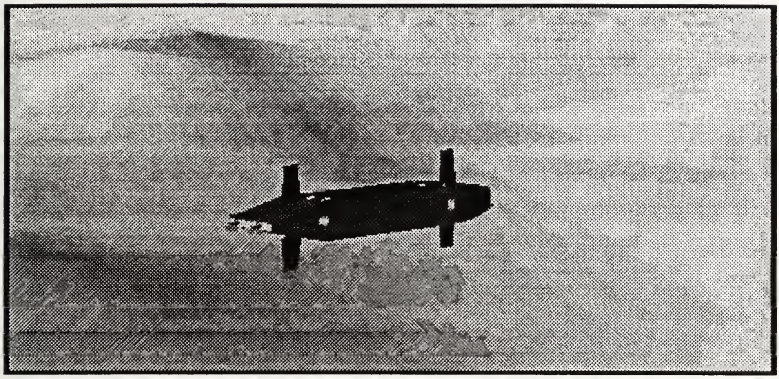


Figure 2.7 A graphics simulator depicting the NPS AUV in Monterey Bay

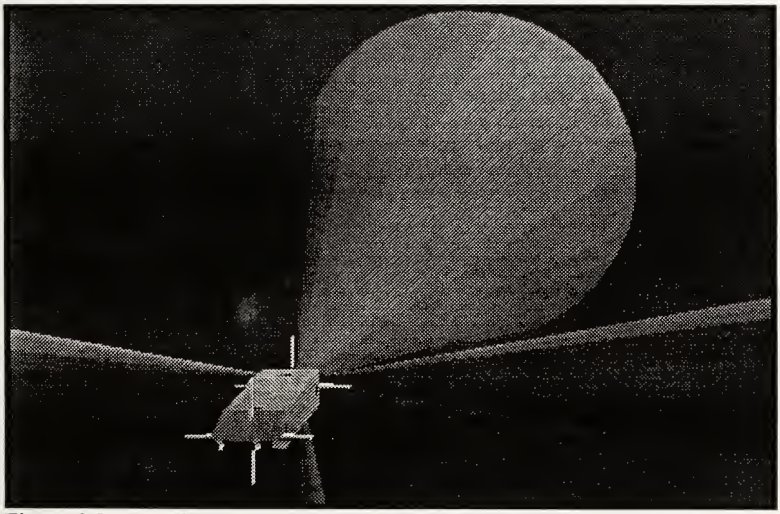


Figure 2.8 Graphics simulation for NPS AUV sonar visualization

flexibly adjusts to changing tactical situations, such as detection of an unexpected obstacle or appearance of a hostile submarine. Search techniques can be used to map

minefields. Expert systems can process sonar data and perform object recognition and classification. Neural networks can be used for fault diagnosis and mechanical system control.

Systems integration is obviously a key factor in the construction of a vehicle with so many different components. There are a large number of hardware and software systems that mutually depend upon each other for successful operation. Similarly, there are a large number of people working on new designs in each of these areas. The requirement to build a robust and independent system means that reliability, redundancy and interoperability must be thoroughly considered during each phase of design and construction.

D. THE FUTURE OF NAVAL AUVs

Potential AUV military applications are limited only by the imagination. Mine detection and minefield mapping might be completely performed by single or multiple AUVs in direct support of an independent submarine or surface ship. Multiple AUVs could quickly clear transit lanes for fleet deployment or amphibious assault. Dropping an AUV from an aircraft could be a quick way to initiate harbor or choke point surveillance, positioning the AUV to act as a "bell ringer" to warn when hostile ships get underway. AUVs have the ability to conduct bottom search in any sea state without requiring a controlling vessel to remain on station. This capability would keep critical operations such as high-value object searches or amphibious landing preparations in progress despite difficult weather conditions. Round-trip or one-way delivery of underwater sensors and weapons becomes much less hazardous. An AUV configured as an artificial target would be an intelligent and realistic adversary during antisubmarine warfare exercises. Software control of autonomous vehicles permits relatively rapid and inexpensive upgrades to quickly adapt to changing enemy capabilities and new mission requirements. Finally, the low radiated noise and small active sonar return of an AUV should result in unmatched stealth, possibly leading to new fleet missions that are not currently feasible.

Much exciting work is in store for future AUV research and development. Video camera recording, real-time vision processing and automatic image interpretation are possible. Green laser (543 nanometer) range-finding is already operational on the Monterey Bay Aquarium Research Institute ROV *Ventana* (Davis 91). High energy-density power sources require continued development in order to take full advantage of AUV capabilities. High performance sonar modifications are being investigated to match AUV space and mission requirements. Modular connections for robot arm manipulators, replaceable packages and deliverable payloads need to be designed for flexible support of all potential mission requirements. Improved computer architectures such as parallel processing transputers will allow simultaneous accomplishment of many tasks in real time, as well as extend the autonomy and artificial intelligence capabilities of these vehicles. High-bandwidth acoustic modems can be used for rapid remote communications with an operating AUV. Battle group commanders should have the ability to receive remote sensor reports via radio uplinks from AUVs on station and send mission commands in return.

Intelligent mobile robots will be performing many missions for the U.S. Navy in the near future. Their employment as extensions of our ships, submarines and aircraft will become commonplace. The imaginations, technical prowess and operational experience of officers and faculty at NPS are making this future a reality.

In undersea warfare, silence and nondetectability are the most important factors in achieving stealth and tactical advantage. The military significance of autonomous underwater vehicles was emphasized during a recent visit to NPS by a Soviet naval delegation. A small group of senior Soviet officers was shown the NPS AUV. The group looked and listened politely. The Soviet admiral asked only one question: "Is it quiet?"

Indeed it is.

III. INTEGRATED SIMULATION FOR RAPID AUV DEVELOPMENT

A. ABSTRACT

The development and testing of Autonomous Underwater Vehicle (AUV) hardware and software is greatly complicated by vehicle inaccessibility during operation. Integrated simulation remotely links vehicle components and support equipment with graphics simulation workstations, allowing complete real-time, pre-mission, pseudo-mission and post-mission visualization and analysis in the lab environment. Integrated simulator testing of AUV software and hardware is a broad and versatile method that supports rapid diagnosis and robust correction of system faults.

Pre-mission simulator AUV testing permits experimental evaluation of developmental software. Pseudo-mission simulator testing of AUV processes employs an identical laboratory microprocessor or remote communication with a testbench-mounted operating AUV, permitting end-to-end testing of all software and hardware. Post-mission simulator playback of recorded telemetry, sensor data and system state transitions supports in-depth reenactment, playback and analysis of in-water operational results.

High-resolution three-dimensional graphics workstations can provide real-time representations of vehicle dynamics, control system behavior, mission execution, sonar processing and object classification. Use of well-defined, user-readable mission log files as the data transfer mechanism allows consistent and repeatable simulation of all AUV operations. Examples of integrated simulation are provided using the Naval Postgraduate School (NPS) AUV, an eight foot, 387-pound untethered robot submarine designed for research in adaptive control, mission planning, mission execution, and post-mission data analysis.

The flexibility, connectivity and versatility provided by this approach enables sophisticated visualization and analysis of all aspects of AUV development. Integrated

simulator networking is recommended as a fundamental requirement for comprehensive and rapid AUV research and development.

B. INTRODUCTION

1. Problem Statement

Designing, building and testing an Autonomous Underwater Vehicle (AUV) is difficult. Unlike most other mobile robots, AUVs must operate unattended and uncontrolled in a remote and unforgiving environment. Inaccessibility greatly complicates evaluation, diagnosis and correction of AUV system faults. In order to ensure complete reliability, AUV software and hardware need to be fully tested in the laboratory before operational deployment. Such important testing requirements cannot be met using only a standalone AUV.

2. Motivation

The principal motivation driving the development of an AUV integrated simulator is to meet the research needs of the large academic group working on the Naval Postgraduate School (NPS) AUV. Students and professors have diverse research goals that are often forced to compete for access to vehicle system software and limited pool test time. The need to use operational software running on actual NPS AUV hardware is a particularly important requirement. Lack of accessibility to the NPS AUV in a distributed laboratory environment has occasionally prevented porting new software applications into the vehicle. Pre-mission validation of vehicle systems response to new software has been similarly limited in scope, resulting in several operational test failures and frustrating delays in development.

The integrated simulation approach has great value and general applicability. The NPS AUV Integrated Simulator has been designed to support complete scientific visualization of actual NPS AUV vehicle performance. The lessons learned while building this integrated simulator have proven that distributed research can be effectively accomplished when proper network connections and data-passing mechanisms are provided.

3. Definition and Objectives of Integrated Simulation

Integrated simulation is defined as the effective networking of a three-dimensional graphical simulation workstation with an AUV microprocessor, appropriate support equipment and all software development workstations. Integrated simulation allows coordinated utilization of computer resources for maximum realism and effectiveness. The purpose of this paper is to demonstrate the use of integrated simulation as an essential approach for rapidly designing, developing and evaluating AUVs.

An AUV integrated simulator remotely links vehicle components and support equipment with graphics simulation workstations. Networking allows complete pre-mission, pseudo-mission and post-mission visualization and analysis in a real-time lab environment. Complete integrated simulator testing of software and hardware supports prompt diagnosis and robust correction of system faults. Figure 3.1 shows connectivity for a sample AUV integrated simulator network and primary components.

Pre-mission simulator testing of AUV software permits experimentation and preliminary evaluation of developmental software. Pseudo-mission testing using an identical laboratory microprocessor or remote communication with an actual AUV permits end-to-end testing of all AUV software and hardware. Post-mission simulator playback of recorded telemetry, sonar sensor data and system state transitions supports in-depth reenactment, playback and analysis of actual operational results.

4. Previous Work

Several graphics simulators have been previously developed at NPS to support AUV research. These simulators all operate on Silicon Graphics Inc. Iris graphics workstations. Seow Meng Ong developed a simulator that remotely networks an Iris workstation with a Symbolics Inc. Lisp machine for real-time communication by mission planning and path planning software (Ong 90) (Zyda 90). CDR Thomas A. Jurewicz USN developed a real-time NPS AUV simulator that featured a complete hydrodynamics model and bathymetric survey terrain data of Monterey Bay

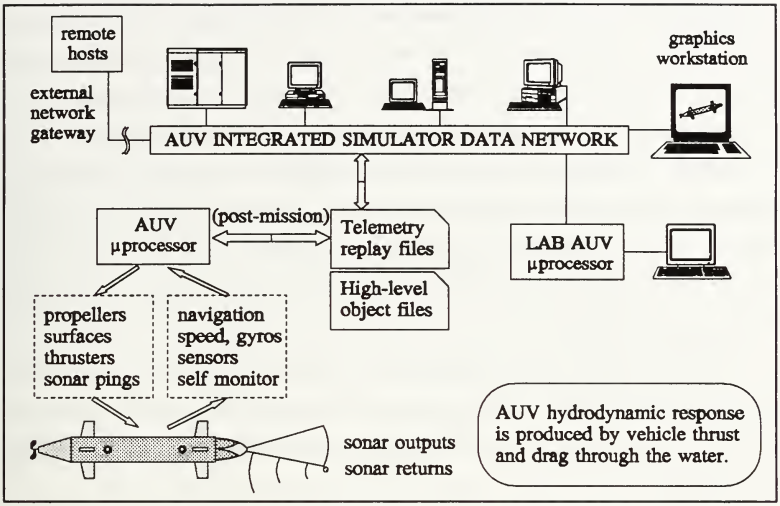


Figure 3.1 Integrated simulator network physical connectivity

(Jurewicz 91) (Zyda 91). CDR Charles A. Floyd USN extended the Jurewicz simulator to demonstrate sonar detection and collision avoidance software (Floyd 91) (Floyd Kanayama Magrino 91). MAJ Ronald B. Byrnes USA and LCDR David L. MacPherson USN utilized the network capabilities of the Ong simulator to visually compare hierarchical and subsumption software architectures for AUV control (Byrnes 92).

Other less complex simulation methods have also been used for NPS AUV development. Most NPS AUV control system theses have analyzed vehicle performance parameters individually using mathematics support packages such as MATLAB (Mathworks 89), forcing researchers to visually correlate numerous two-dimensional plots of telemetry data in order to interpret test results.

Other underwater vehicle projects have also used offline graphics simulation as a design tool. As an example, C.S. Draper Laboratories has a large and sophisticated simulator that supports the development of the Defense Advanced

Research Projects Agency (DARPA) Unmanned Underwater Vehicle (UUV). This simulator employs sophisticated computer models of hydrodynamic characteristics and individual physical component responses. Mission software is loaded on a separate mainframe to emulate vehicle multiprocessor response. The simulator does not incorporate actual DARPA UUV multiprocessor hardware or allow direct playback of UUV system and sensor data collected in the water. However the many capabilities of this powerful support simulator have significantly contributed to the reliability of the DARPA UUV, allowing successful and rapid progress along an ambitious development schedule (Pappas 91) (Hale 91).

All of these simulation approaches successfully demonstrate the concepts they are intended to evaluate. However, none of these simulators were designed to use actual vehicle hardware or to provide general extendability to support every aspect of AUV research.

C. AUV DESIGN AND DEVELOPMENT CONSIDERATIONS

AUVs are complex systems. A number of design and employment criteria unique to AUVs must be considered when determining integrated simulator specifications.

1. AUV Inaccessibility During Operation

The development and testing of AUV hardware and software is greatly complicated by vehicle inaccessibility during operation. AUVs are designed to operate with complete independence in an environment that makes communication and monitoring difficult. Vehicle independence design constraints leave operators unable to monitor performance, diagnose problems or override failures. This inaccessibility is perhaps the biggest liability inherent in AUV testing since it can easily lead to catastrophic failure and vehicle loss. Even when supervisory control is possible through use of a tether or underwater communications, underwater vehicle systems must be robust enough to recover and return in the event of system failures combined with communication loss. Integrated simulation can fully test fault tolerance and

emergency recovery procedures of an AUV prior to risking loss of communications during independent operation.

2. Reliability is Paramount

Loss of an AUV due to internal failure or inability to cope with an unpredictable environment is unacceptable due to the current high cost of AUV construction and support. Furthermore if an AUV is employed in military missions such as submarine support or minefield search, human lives and operational success may depend on complete vehicle reliability. Thus the principal requirement for any AUV is that the vehicle operates dependably in all possible scenarios and under all possible failure conditions. Pre-mission verification of proper AUV performance using an integrated simulator is the only way to ensure complete vehicle integrity and verify strict reliability requirements for all software and hardware components.

3. Wide Variety of Software Process Types

The highly complex behaviors expected of AUVs are only possible when numerous software modules are written to handle functions such as path planning, sonar interpretation, mission control etc. Such software programs can be considered artificial intelligence (AI) applications in that human intelligence might otherwise be required to perform these challenging tasks. It is important that these high-level software modules are able to fully interact with each other for proper execution and evaluation. However such interaction is difficult when the researchers developing software are distributed over a network. Integrated simulation provides full connectivity between research software modules and the AUV microprocessor. Integrated simulation also provides data-passing mechanisms that permit interprocess communication regardless of the various host operating systems or programming languages used.

D. INTEGRATED SIMULATOR SOFTWARE ARCHITECTURE

1. Software Engineering Considerations

Proper design of an AUV integrated simulator addresses many requirements including repeatability, flexibility, cost-effectiveness, portability, maintainability, future growth potential and ability to upgrade. These goals can be met by following fundamental software engineering principles such as clearly defining software module specifications and functional descriptions. Formally defined data dictionary entries, data structures and spatial coordinate systems are also important. Specifications must be flexible enough to support future improvements and comprehensible enough to be rigorously followed. Frequent and thorough communication and cooperation among project members is important in order to establish formal project standards and ensure long-term success.

2. Integrated Simulator Software Architecture Requirements

Integrated simulator software must perform a large number of tasks. The AUV must be modeled using some simulated components (e.g. control surfaces, propellers, gyrocompass) together with actual running AUV mission software. Vehicle physical motion and behavior can be provided by the state equations of a dynamic response model. The world model needs to include stationary obstacles, mobile objects and the sensor interactions expected to occur as the vehicle probes the external environment model. Developmental AUV processes that have not yet been ported into the vehicle mission software need to have some way of interfacing with both the AUV microprocessor and the simulation. Operating system and programming language incompatibilities should not be an impediment to AUV software developers. Finally and most importantly, a powerful graphics workstation must render an external view of the simulated world in three dimensions with full functionality and real-time response.

3. Simulated and Actual Components

Maximum simulation realism is provided when actual AUV components are tested end-to-end in the laboratory. For example, an AUV might be fixed in place on

blocks in a test tank while a test mission was conducted. Proper activation of sonar, rudders, diving planes and propellers would provide positive indications of correct performance. It is interesting to note that networking a test-tank AUV to a graphical simulator can give evaluators real-time insight into what the vehicle "thinks" it is doing. However, if a laboratory AUV microprocessor is used instead of the actual vehicle, the missing vehicle physical components must be separately simulated. Such simulation is accomplished by modular substitution of mathematical models for the missing physical components. A particular benefit of this approach is that AUV software testing is freed from direct interaction with the actual AUV, since the vehicle might be operating, undergoing repairs or otherwise inaccessible. The logical relationships between AUV, simulator, laboratory development network and real-world environment are shown in Figures 3.2 and 3.3.

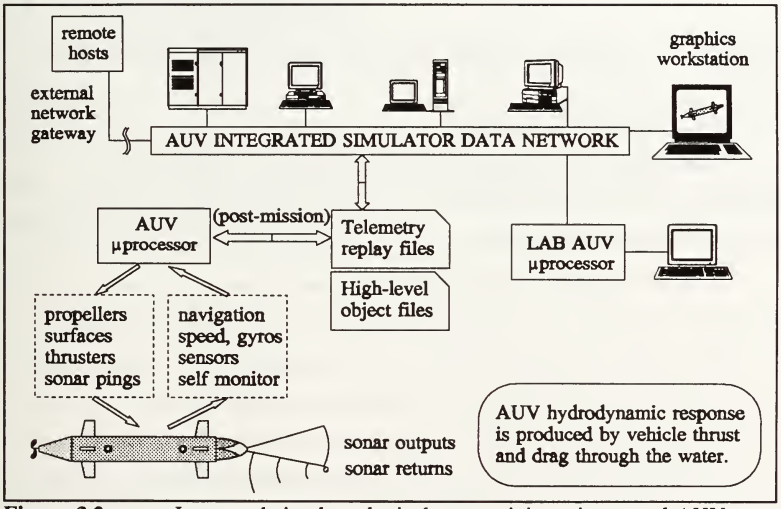


Figure 3.2 Integrated simulator logical connectivity using actual AUV

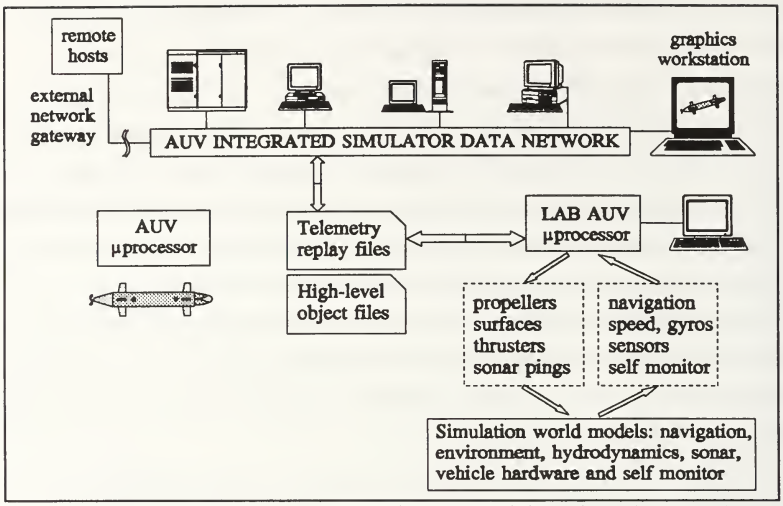


Figure 3.3 Integrated simulator logical connectivity using laboratory AUV

4. Data Transfer Mechanisms

Data transfer mechanisms are a critical component of interprocess communication. Two file types and two data transfer mechanisms are considered: telemetry replay files, high-level object files, remote file transfer and stream sockets.

In order to portray and replay AUV behavior, telemetry recorded by the vehicle must be readable by the integrated simulator. Typically such data includes vehicle position, vehicle orientation, linear and rotational velocities or accelerations, sensor data and vehicle state information, all repeated at a high data rate. Telemetry replay files can be saved by an AUV for post-mission upload or transmitted during operation. These files can also be read (with effort) by human operators, or ported as input to mathematics support packages for selective analysis of system parameters. However, Figure 3.4 illustrates the difficulty in portraying three-dimensional AUV track data using two-dimensional time-versus-z and x-versus-y plots (Compton 92).

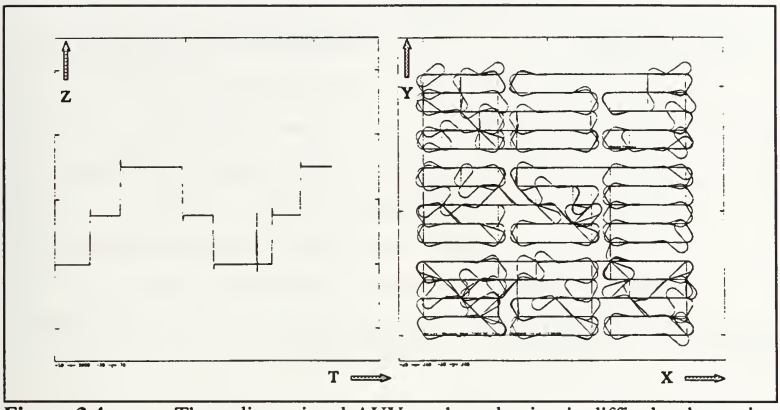


Figure 3.4 Three-dimensional AUV track evaluation is difficult when using multiple two-dimensional plots

Use of well-defined and consistent telemetry replay files allows repeatable simulation of all AUV missions. Telemetry replay files are also a convenient method for new mission software to record primary aspects of AUV behavior during standalone testing for later visualization on the integrated simulator. Figure 3.5 shows an example telemetry replay file format.

```
<time>                          ; telemetry data point time
<x>   <y>   <z>                 ; vehicle estimate of position
<φ>   <θ>   <ψ>                 ; measured 3D orientation
<p>   <q>   <r>                 ; navigation system velocities
<Δ dive>    <Δ rudder>          ; plane surface positions
<rpm>       <log speed>         ; ordered and measured speed
<sensor data fields>            ; all possible sensor returns
<wildcard>                      ; extra slot for mission-dependent use
```

Figure 3.5 Example telemetry replay file format

High-level object files allow communication of symbolic data such as position of objects, object classification, operator instructions and interprocess commands. Keeping such data in plain text makes them readable by human operators, individual AUV software processes and the integrated simulator. Optional time parameters on each command line allow high-level object files to supplement telemetry replay files for synchronized real-time playback. This combination of telemetry replay files and high-level object files allows simple and effective communication of all possible types of AUV information. An example high-level object file format is shown in Figure 3.6.

File transfer is the fastest and easiest way to record and communicate large amounts of data over a distributed research network. An integrated simulator network must be able to transfer telemetry replay and high-level object files between all network nodes.

Once a file transfer capability has been established, stream sockets can be implemented if transfer of individual data packets is desired (Barrow 88). Stream sockets can connect all processors on an integrated simulator network, allowing direct interprocess communication, near real-time data transfer and better evaluation of multiple process interaction.

```
Environment     "worldfilename"                     ; change default world file
AUV       <x>   <y>   <z>                           ; AUV initial position
Point     <x>   <y>   <z>                           ; Point position coordinates
Segment   <x1>  <y1>  <z1>  <x2>  <y2>  <z2>  ; endpoint coordinates
Wall      <x1>  <y1>  <z1>  <x2>  <y2>  <z2>  ; opposite corners
Cylinder  <x>   <y>   <z>   <r>   <h>
Mine      <x>   <y>   <z>   <scale>           [time <t>] ; time optional
Ship      <x>   <y>   <z>   <scale>           [time <t>]
Object    "filename"        <x>   <y>   <z>   [time <t>]
Message   [time <t>]        ... free format text here ...
; messages can be mission log outputs or interprocess communication
```

Figure 3.6 Example high-level object file format

5. Distributed Artificial Intelligence Considerations

A large number of interrelated AI software processes are required for an AUV to competently perform the many behaviors required of an independent submersible. In order to keep up with demanding mission requirements, these processes must be capable of performing in real time and in parallel. Similar real-time and parallel processing support will be necessary for a graphics workstation to provide correspondingly realistic playback and interaction.

Interprocess communication and real-time process interaction are usually difficult to implement, especially if multiple user, multiple programming language or multiple operating system bottlenecks exist. The data transfer mechanisms described previously permit complete interaction among dissimilar distributed AI applications, regardless of whether these applications are internal or external to the AUV. This straightforward approach allows complete user and simulator accessibility to intermediate process outputs.

E. THREE-DIMENSIONAL GRAPHICS SIMULATION

High-resolution three-dimensional graphics workstations provide realistic representations of vehicle dynamics, control system behavior, mission execution, sonar processing and object classification.

1. Realistic Object Rendering and Real-Time Motion

The primary graphics requirement for an integrated simulator is realistic rendering and movement of virtual objects in real time. This capability is essential for visualizing an AUV's interaction with an underwater world in order to fully evaluate the proper operation of complex AUV software and hardware. Numerous graphics techniques can be used to provide a believable graphics display, ranging from drawing simple polygons to overlaying complex textures. Realistic portrayal of all objects in an underwater world allows intuitive and thorough analysis of large amounts of AUV data.

Maintaining a real-time playback capability is important for realistically rendering AUV interaction with physical objects. The graphics simulator program must be able to quickly refresh complex screens in order to visually present large amounts of data. Local empirical studies show that a 6 Hz screen update rate and input device response loop are the minimum requirements for simulator screen motion to appear smooth and realistic during operator interaction. Frame rates of 20-30 Hz may be needed for realistic illusion of rapid motion (Brooks 88). Speed can be increased and graphics pipeline loading reduced through simplified object geometry, simplification of lighting models, simulator source code optimization and graphics performance tuning techniques.

2. Physical Modeling

All AUV-related physical processes can be mathematically modeled with a high degree of accuracy. Vehicle physical response can be predicted using state equations, positional constraints, inverse kinematics and dynamics (Jurewicz 90) (Thalmann 90) (Badler 91). Sonar acoustic behavior can be modeled with increasingly complex levels of detail in order to meet both realism and system playback

requirements (Etter 91). Individual AUV hardware components can be simulated using control system models of transient and steady-state response. Object motion is adequately modeled using simple kinematics. Object positions can be easily updated whenever more recent correlated sonar data becomes available. In general, physical modeling is less processor-intensive than graphics rendering and adds no apparent overhead to graphics workstation response when properly parallelized (Jurewicz 90) (Zyda 91).

3. Sonar and Sensor Visualization

Sonar data is often difficult to visualize since acoustic beam and ray path behavior is very different from our vision-based perceptual expectations. Sonar remains the primary sensor used for intermediate and long range underwater detection. Sonar can also be quite effective when used for short range detection, object feature extraction or measurement of object characteristics such as doppler or frequency response. Color graphics visualization can portray the real-time behavior of sonar beams in three dimensions, allowing AUV designers to troubleshoot complex problems, optimize vehicle sensor performance and better understand how an AUV is interacting with the environment (Brutzman Compton 92) (Brutzman 92) (Compton 91). Other types of sensors such as laser rangefinders can also be displayed. Sensor visualization capabilities are valuable features for an integrated simulator.

F. INTEGRATED SIMULATOR HARDWARE ARCHITECTURE

1. Workstation Compatibility

There are surprisingly few hardware constraints on the individual workstations making up the distributed network portion of an integrated simulator. A variety of normally incompatible operating systems and programming environments may be used as long as network connections provide a open data transfer path. Even application source code may be in a language foreign to the AUV. For example, a high-level language (e.g. Lisp, CLIPS or Prolog) may be used for rapid prototyping

and initial development. Testing is then accomplished using interprocess communication and real-time data transfer of high-level object file information with the AUV. After initial process testing is complete, working high-level language code can be translated and ported into the native language of the AUV (e.g. ANSI C). This open architecture approach allows great flexibility and maximum use of available resources.

2. External Network Connectivity

Fully networked connections between all major support components of the AUV is essential to provide a responsive research environment. Additional external network connections will further extend AUV integrated simulator capabilities. For example, laboratory data transfers over a wide-area network or Internet allow joint AUV research over long distances. For another example, actual telemetry replay files can be transferred from a moored AUV via modem or radio link for immediate remote replay, analysis and verification. Such capabilities are particularly important when an AUV is deployed at great distances from support laboratories and immediate analysis of collected information is necessary.

G. IMPLEMENTATION, EVALUATION AND EXPERIMENTAL RESULTS

An integrated simulator has been implemented for the NPS AUV (Brutzman 92). This section describes the primary components and key features of the NPS AUV Integrated Simulator.

1. NPS AUV Vehicle Description and Sonar Characteristics

Naval officers and civilian scientists at NPS are conducting active research using an AUV designed and constructed at the school. The NPS AUV is used for basic research and thesis work in control systems technology, artificial intelligence, scientific visualization and systems integration. Specific NPS AUV project objectives include the study of mission planning, navigation, collision avoidance, real-time mission control, replanning, object recognition, vehicle dynamic motion control, and post-mission data analysis (Healey 91) (Brutzman Compton 91).

The NPS AUV is eight feet long and neutrally buoyant, displacing 387 pounds with overall size and shape comparable to a small dolphin. Current vehicle endurance is two to three hours. Maximum speed of the NPS AUV is about two knots. The NPS AUV turning diameter is under three body lengths, designed to be ideal for maneuvering in the large NPS swimming pool. The NPS pool allows precise testing in a quiet, controlled environment. Open-ocean testing is feasible but is being reserved for a more robust follow-on vehicle. Video clips showing normal NPS AUV operation are available in (Brutzman, Floyd, Whalen 92) and (Brutzman 92).

The primary components of the NPS AUV are an aluminum hull, fiberglass sonar dome, four high-frequency directional sonar transducers, twin counter-rotating four-inch propellers, lead-acid batteries, eight plane surfaces, and a Gespac computer running a Motorola 68030 processor with a 2 MB RAM card under the OS-9 operating system. Figure 3.7 shows a general schematic of the NPS AUV.

Four PSA-900 Programmable Sonar Altimeters made by Datasonics Inc. are orthogonally fixed in the nose of the NPS AUV pointing directly ahead, downward and to port and starboard. These transducers are fixed frequency and ultrasonic, each at approximately 200 Khz. Sonar range gate is selectable at 30 m or 300 m, and pulse length is 350 μs. Normal pulse repetition rate is 10 Hz. Sonar beamwidth is seven degrees and range resolution is 1 cm at 30 m.

2. NPS AUV Integrated Simulator

The NPS AUV Integrated Simulator has been developed to support NPS AUV research and demonstrate each of the concepts described in this paper (Brutzman 92).

High-level NPS AUV software processes are initially developed and tested on the Unix-based computer science department network. These processes can now be ported, compiled, linked and loaded on a Gespac VME-bus 68020 or 68030 microprocessor running under the OS-9 operating system. Gespac microprocessors are used both on the NPS AUV and on a separate networked laboratory AUV. The laboratory AUV includes 68020 microprocessor and I/O cards, a monitor terminal,

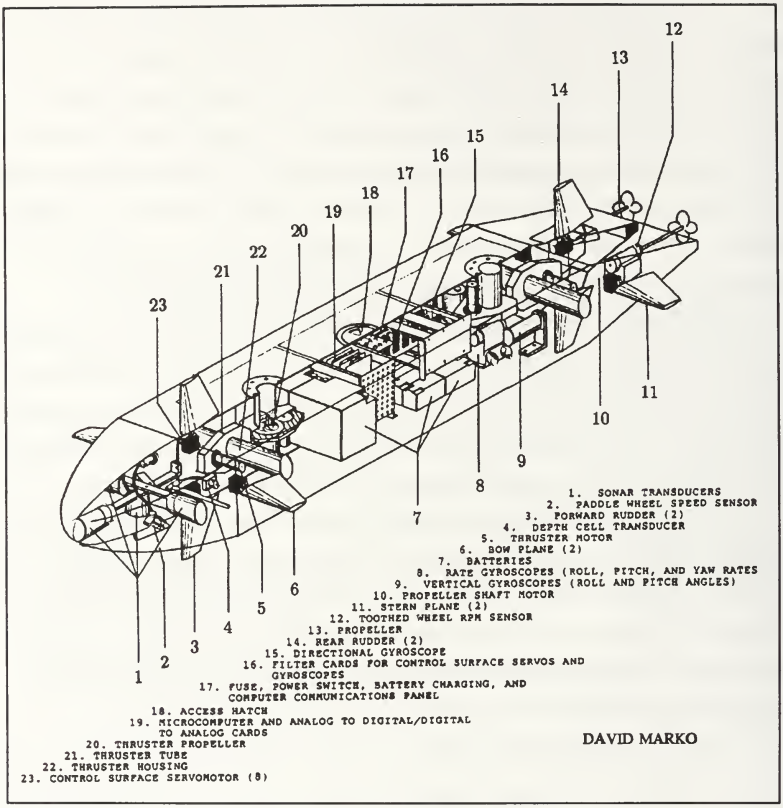


Figure 3.7 General schematic of NPS AUV to scale

Ethernet network connections and a networked IBM-compatible support PC that includes an OS-9 "C" language cross-compiler. The laboratory AUV also has additional hardware card slots in order to test new hardware components and new vehicle software.

Graphics simulation using the NPS AUV Integrated Simulator is just beginning to be used for laboratory evaluation of software that will run in the AUV

proper. New hardware and software can be rapidly tested prior to installation and operation in the NPS AUV, minimizing vehicle risk while saving time and money. Replays of actual data recorded by the NPS AUV can be used for visualization of remote environments and detailed post-mission data analysis. Connection of software development workstations with the NPS AUV Integrated Simulator accelerates the operational deployment of high-level mission software.

The NPS AUV Integrated Simulator control panel has been written using NPS Panel Designer software, making it quickly modifiable and extendable (King Prevatt 91). The graphic simulator user interface permits precise control of viewpoint and reference point, lighting and rendering functions, object positions, real-time mode, high-level object file recall, individual object control and playback of telemetry replay files. The NPS AUV Integrated Simulator control panel is shown in Figure 3.8.

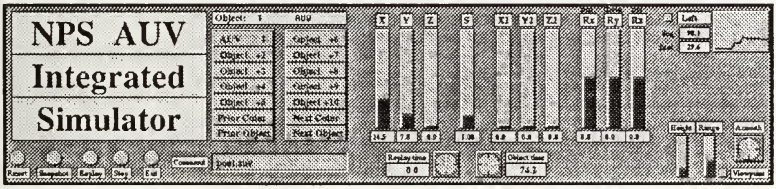


Figure 3.8 Control panel for the NPS AUV Integrated Simulator

Because the AUV hull shape is similar to the original Swimmer Delivery Vehicle used by U.S. Navy SEAL teams, a sophisticated mathematical model is already available for simulator use to accurately recreate vehicle dynamic motion and response characteristics (Jurewicz 90). The hydrodynamics model state equations contain approximately 120 coefficients that continue to be improved and verified by pool testing and ongoing thesis work.

In order to display a variety of sonar data, multiple objects can be displayed on the graphics workstation. Implemented object primitives include AUV, point, line, wall, mine and cylinder. These objects can be graphically displayed simultaneously

with original telemetry replay data in order to analytically visualize the validity and usefulness of various sonar classification techniques. Objects can be independently manipulated and positioned. An optional time slot for each object allows them to appear only when appropriate during synchronized playback of telemetry files. Additional advanced graphics techniques can quickly be added to the baseline graphics simulation program.

3. Silicon Graphics IRIS Workstation Capabilities

The NPS AUV Integrated Simulator uses a Silicon Graphics Inc. Iris 4D/240VGX. This graphics workstation has 48 bit color, 24 bit Z-buffering and four parallel 25 Mhz 20 MIPS processors that together can process 1 M vectors, 1.1 M triangles or 180 K polygons per second (Gorey 91). Other slower IRIS workstations are also available for use, including a remote workstation in the Mechanical Engineering Department adjacent to the NPS AUV support laboratory. All graphics workstations are connected by local or wide area networks. Real-time playback of telemetry data is automatically adjusted to take maximum advantage of the current graphics workstation processing power, producing realistic screen displays regardless of which model graphics workstation is used.

4. Laboratory AUV Simulation

A primary objective of integrated simulation is to run operational software on a laboratory version of the AUV microprocessor. The NPS AUV Integrated Simulator includes an identical Gespac computer running a Motorola 68030 under the OS-9 operating system. Added to this computer are interface cards for a VT220 monitor and keyboard for external control, serial connection to a PC, and Ethernet connection to the NPS computer science department network. Full connectivity is thus provided to all developmental workstations of interest as well as the Campus Wide Network and Internet. Since OS-9 is a multiprocess real-time system, multiple users can access the Gespac AUV microprocessor simultaneously.

Unmodified operational NPS AUV software is able to run successfully on the laboratory AUV microprocessor, and telemetry data files are properly saved during

each run. Telemetry data files have been successfully transferred over the network and played back on the Iris graphics workstation. Although missing NPS AUV hardware such as sonar and plane surface response has not yet been simulated, successful visualization of the laboratory test runs has proven the feasibility of the integrated simulation approach. Functional AUV software and hardware is now directly available to all NPS AUV researchers for experimentation and evaluation prior to in-water testing.

H. ADDITIONAL APPLICATIONS

Several applications were implemented concurrently with the NPS AUV Integrated Simulator that successfully demonstrate the usefulness of integrated simulation in support of high-level AUV-related AI research.

1. Sonar Classification Application

The NPS AUV Sonar Classification System uses outputs from simple active sonars to classify detected underwater objects (Brutzman Compton Kanayama 92) (Brutzman 92). Figure 3.9 shows sample sonar classifications in the NPS pool displayed using the NPS AUV Integrated Simulator. Scientific visualization techniques permitted rapid and precise development of geometric analysis techniques and classification heuristics, resulting in successful completion of the NPS AUV Sonar Classification System.

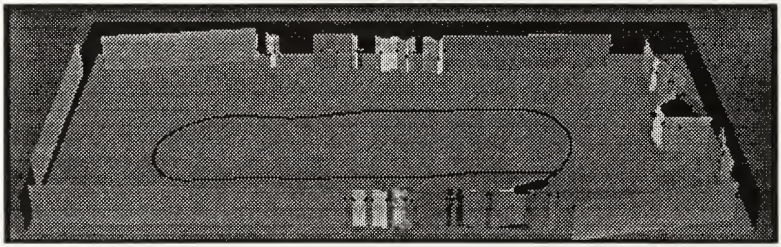


Figure 3.9 Integrated simulator screen display of the NPS pool, AUV track and all active sonar classifications

2. Circle World Path Planning Application

Optimal path planning is an important area of AUV research. Displaying and viewing paths and obstacles without restrictions allows the algorithm designer to evaluate his results in the most comprehensive and challenging manner possible. Additionally, subtle difficulties that might be obscured by two-dimensional projections are clearer and easier to evaluate when shown in three-dimensions. A circle world path planner has been developed that finds shortest paths around circular or cylindrical obstacles (Brutzman 92). Figure 3.10 shows how shortest path planning results can be portrayed in three dimensions using the NPS AUV Integrated Simulator.

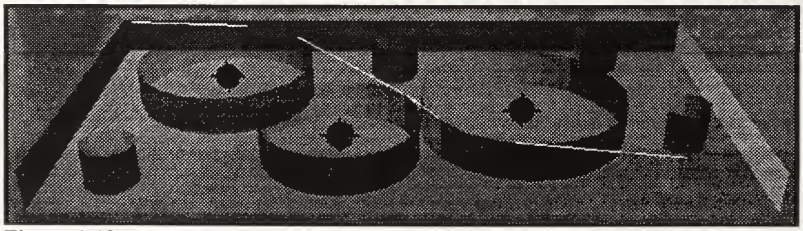


Figure 3.10 Integrated simulator three-dimensional representation of circle world obstacles and shortest path in the NPS pool

3. Minefield Search Application

Another application benefiting from integrated simulation is an AUV minefield search planner (Compton 92). A three-dimensional open-ocean minefield model is optimally searched and mapped using a dynamic search strategy. AUV search track and vehicle posture are recorded in a simplified telemetry replay file, while waypoint objectives and detected mines are recorded in a separate high-level object file. Synchronized playback of these files allows complete visualization of the complex path taken by the AUV as well as the numerous objects detected, shown in Figure 3.11. Note that vehicle track is much easier to visualize than in Figure 3.4, particularly since the simulator user's viewpoint can be panned over and around the track data.

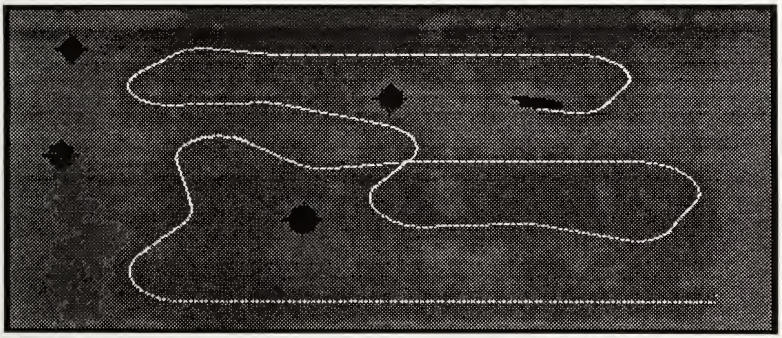


Figure 3.11 Integrated simulation display of AUV minefield search

I. ADDITIONAL APPLICABILITY, LIMITATIONS AND FUTURE WORK

1. Comparison of Theoretical and Empirical Data

Three-dimensional visualization techniques are well suited for making meaningful comparisons between large abstract data sets. Such comparisons can significantly aid the operator in evaluating small errors in mathematical models or system control software. For example, predicted vehicle track for a given control

systems algorithm could be spatially superimposed over actual test track data. Coefficients in the prediction model can then be incrementally adjusted until theoretical behavior matches actual performance. A similar visualization approach was used with great success while determining precise heuristics for object classification using recorded active sonar data (Brutzman Compton Kanayama 92) (Brutzman 92). Direct comparison of theoretical and empirical data is a powerful diagnostic tool that can be used to improve theoretical formulations as well as vehicle implementations.

2. Limitations to Integrated Simulation

The primary limitation on integrated simulation realism is graphics workstation speed and capability. Many graphics workstations can generate photorealistic images but are unable to rapidly reproduce a series of images in real time. The competing requirements between rendering accuracy and adequate frame rate will always require design tradeoffs by the graphics programmer. Silicon Graphics Inc. workstations use the GL Graphics Library, which is a good graphics programming choice due to the numerous graphics techniques provided, code optimization, portability to other platforms and open licensing availability. Graphics workstation capabilities are probably the most critical consideration in integrated simulator design.

The local area network (LAN) used to connect integrated simulator nodes should be reliable, have adequate throughput and allow addition or removal of nodes with little difficulty. Ethernet-based LANs are adequate for NPS AUV Integrated Simulator requirements and also provide gateway connectivity to Internet. It should be noted that under most network protocols socket stream packet delivery order is not guaranteed and timing of packet delivery is somewhat unpredictable. Processes that use socket stream data should be flexible and not tied to hard real-time requirements.

Computer security is a consideration if sensor data or mission software is proprietary or classified. The use of plain text for telemetry replay files and high-level object files permits the use of encryption protocols during transfer. Encrypting files is a simple technique that imposes minimal processing overhead. Individual nodes on

the integrated simulator network will require standard security precautions against unauthorized remote access.

3. Future Use of Integrated Simulation

Integrated simulation provides development benefits to all types of remote vehicles, regardless of whether a communications tether is present or remote control by human operators is required. Integrated simulation not only solves a number of the problems that degrade robot implementation, but also provides tools to work on practical system engineering and integration problems that previously were too difficult to address. The authors hope that widespread incorporation of integrated simulation techniques will improve the accessibility, intelligibility and progress rate of mobile robot research.

J. CONCLUSIONS

Integrated simulation allows all AUV systems to be tested in a timely and complete manner. The flexibility and connectivity provided by this approach enables sophisticated visualization and complete analysis of all aspects of AUV development. Integrated simulator networking is recommended as a fundamental requirement for comprehensive and rapid AUV research and development.

IV. NPS AUV INTEGRATED SIMULATOR DESIGN SPECIFICATIONS

This chapter provides details about design specifications particular to the NPS AUV and the NPS AUV Integrated Simulator. Specifications must include both vehicle and integrated simulator requirements for compatibility. Users and programmers need to comply with or formally improve design specifications in order to maintain forward and backward compatibility throughout the effective lifetime of the NPS AUV research project. A valid and effective data dictionary defined in (Floyd 91) is used for all data types defined in this thesis.

A. NPS AUV ACTIVE SONAR SYSTEM

The current NPS AUV active sonar uses four ultrasonic directional beams pointed ahead, down and 90° to port and starboard. Individual transducers are mounted on adjustable semicircular frames that can allow all beams to be directed forward. However the dimensions of the NPS pool and the 30 m range gate of the transducers make orthogonally oriented transducers an optimal approach for sensing multiple walls and targets simultaneously (Figure 4.1). The NPS AUV active sonar characteristics are further described in Chapter VI and (Floyd 91).

Several problems have handicapped NPS AUV sonar performance. Faults in the signal processing electronics have prevented simultaneous utilization of multiple sonar transducers. Automatic averaging of sonar ranges at the board level has introduced minor range errors and reduced the discrimination capability of the transducers against small targets. The strongly reflective walls and shallow depth of the NPS pool create a high reverberation environment. Despite these significant difficulties, adequate sonar classification results have been obtained inside the NPS pool.

In a real sense, the independence and capabilities of any mobile robot (or even manned submarine) are constrained by the quantity and quality of sensor data available about the external world. Correction of current NPS AUV sonar limitations is essential if more accurate vehicle control and higher level behaviors are to be achieved.

Figure 4.1 NPS AUV sonar beam profiles in the NPS pool

B. NPS POOL COORDINATE SYSTEM

Several theses have graphically modeled the NPS swimming pool or utilized AUV data collected in the NPS pool. Unfortunately most of these efforts are incompatible because no standard pool coordinate system has been established. In order for test results to be understandable and repeatable throughout the life of the NPS AUV program, a standardized NPS Pool Coordinate System is defined here.

Numerous competing criteria were resolved when defining this coordinate system, especially differences between coordinate systems used by graphics simulation programs. Advantages of the NPS Pool Coordinate System are as follows:

- all pool coordinates positive and units in feet
- surface depth z equals zero, increasing depth corresponds to increasing z (indicated on diagram by tail of z-axis arrow)
- NPS AUV data file coordinates become standardized for readability and future reference
- vehicle position and posture terminology are standardized
- right-hand rule relationship between all three axes maintained
- compatible with vehicle coordinate system and Euler angle definitions
- typical start points and normal operator's perspective are near pool origin
- AUV reference point between center of gravity and center of buoyancy
- angle orientations and coordinate positions are directly compatible with the most prevalent robotics conventions, Dr. Kanayama's spatial reasoning function definitions and standard "C" language trigonometric function calls
- vehicle headings are measured in clockwise direction as are conventional compass headings that are familiar to naval officers
- NPS Pool Coordinate System simultaneously combines Cartesian coordinate plane characteristics, Euler angles and right-hand rule, advantages that are not possible with any other spatial representation

Disadvantages of this coordinate system are as follows:

- similarity to Cartesian plane is only evident from a perspective looking up to the pool surface from below, thus axis orientations may initially be counterintuitive

The NPS Pool Coordinate System is shown in Figure 4.2.

Figure 4.2 NPS Pool Coordinate System

C. NPS AUV TELEMETRY REPLAY FILE FORMAT

Telemetry replay files are a critical component of an integrated simulator. These files are used to record in-water test data as well as portray expected vehicle behavior from offline simulation programs. Numerous variant formats exist for NPS AUV telemetry replay files recorded to date. This variability leads to confusion and incompatibility that worsens as an ever-growing number of unique telemetry replay files become available.

The primary reason behind the current plethora of telemetry data file formats is that different NPS AUV evaluation runs tend to test different hardware or software components. Different tests have correspondingly different data logging requirements. A telemetry replay file format standard has to be flexible to support varying data logging requirements. The current NPS AUV telemetry replay file format is defined in Figure 4.3.

NPS AUV telemetry is recorded at a 10 Hz data rate, allowing precise measurements of varying position, posture and sonar values. A single wildcard data slot can take advantage of this rapid update rate to allow recording of multiple test parameters. For example, suppose a special test needed to record both left and right commanded motor RPM. Alternate wildcard slots can be used for the two parameters and a 5 Hz data logging rate for each is maintained. Another method might record three 3-digit numbers by multiplying each by an appropriate powers of ten. For example, the numbers 200, 300 and 400 can be saved as the single wildcard slot value "200,300,400.00", permitting three parameters to be logged at the full 10 Hz data rate. Thus this newly standardized telemetry replay file format has built-in flexibility to allow compatible recording of variable data logging requirements.

```
; Telemetry track data entries are uniquely identified by time value

<time>          = REAL  ; time of sonar ping

<x>, <y>, <z>   = REAL  ; AUV position at time of ping

<φ>, <θ>, <ψ>   = REAL  ; roll, elevation, azimuth angles
                        ; vehicle posture at time of ping

<p>, <q>, <r>   = REAL  ; vehicle posture angular rates of change
                        ; from navigational gyros

<Δ dive>        = REAL  ; Ordered dive plane angle

<Δ rudder>      = REAL  ; Ordered rudder plane angle

<range1>        = REAL  ; Forward transducer range

<range2>        = REAL  ; Left    transducer range

<range3>        = REAL  ; Right   transducer range

<range4>        = REAL  ; Depth   transducer range

<speed>         = REAL  ; output from paddlewheel sensor

<wildcard>+     = REAL  ; available for user-defined data logging
```

Figure 4.3 NPS AUV telemetry replay data file format specification

D. SOFTWARE PROCESS SPECIFICATIONS

Numerous mutually-dependent processes are necessary for an AUV to have adequate capabilities to perform independent missions. Software module specifications need to be standardized and clearly defined in a large multi-year multi-programmer project such as the NPS AUV. Figure 4.4 is a sample format that can be used to record individual process specifications so that NPS AUV programming group members are able to provide proper process inputs and outputs. A group commitment

to defining, following and updating formal specifications is essential for reliable process interaction. Failure to follow formal specifications will inevitably lead to loss of reliability, mission failure and probable vehicle loss.

E. CONCLUSIONS

Many software modules have been written for eventual use by the NPS AUV but very few have been implemented and used on the vehicle. This chapter has presented design specifications and basic software engineering considerations for the NPS AUV that are essential for compatible operation of the NPS AUV Integrated Simulator. Numerous additional requirements have yet to be formally evaluated. A cohesive software engineering approach is needed to formally define system software architecture, process specifications, process data requirements, standardization of process outputs and software version control. Failure to address these requirements will handicap group research and prevent full implementation of NPS AUV mission software.

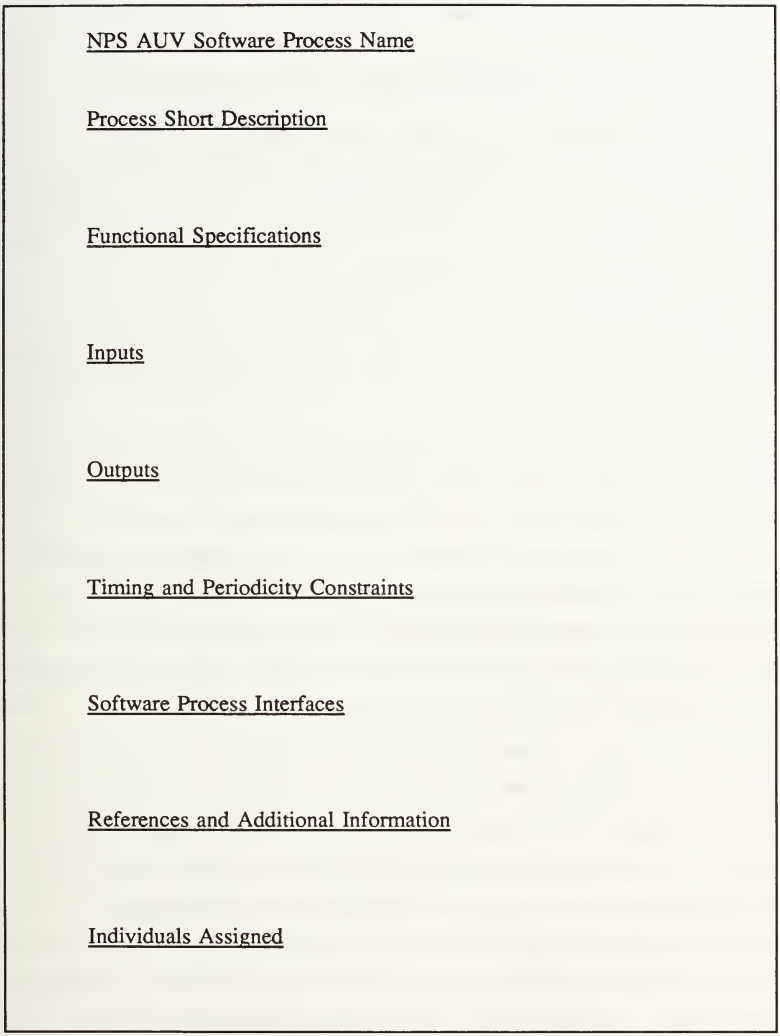


Figure 4.4 NPS AUV software process summary sheet

V. NPS AUV INTEGRATED SIMULATOR DATA NETWORK

A. INTEGRATED SIMULATOR DATA NETWORK OBJECTIVES

The purpose of the NPS AUV Integrated Simulator data network is to connect all workstation and personal computer nodes with the NPS AUV and laboratory AUV microprocessors. This has been accomplished by connecting several local area networks with a wide area network and then adding modems for additional long distance connectivity. This chapter describes how network protocols were chosen and network connections are implemented to support the NPS AUV Integrated Simulator.

B. NETWORK CONNECTIVITY REQUIREMENTS AND ETHERNET

The NPS AUV project has already supported thesis work for dozens of graduate students and will continue to be active for the foreseeable future. The NPS AUV Integrated Simulator data network is designed to maximize accessibility to NPS AUV hardware and software. Network connectivity objectives must address many requirements including compatibility, flexibility, cost-effectiveness, portability, maintainability, future growth potential and ability to upgrade. These criteria are all dependent upon the type of network protocol chosen.

Ethernet is the primary local area network protocol used at NPS. Ethernet is used for the Computer Science Department network in Spanagel Hall and for the NPS Campus-Wide Network. Ethernet is a broadcast bus network with a data transfer capacity of 10 M bps. It is a broadcast network because all transceivers receive every transmission and a bus network because all nodes are joined by a common communications channel. Bridge boxes are used to connect various departmental local area networks to the Campus-Wide Network. These bridges screen unnecessary traffic from transmission to adjacent networks, and also pass copied packets of interest without reproduction of noise or packet collisions.

Protocols describe methods of passing messages, formatting data and handling error conditions. TCP/IP (Transmission Control Protocol/Internet Protocol) is the official name of the Internet Protocol Suite used by Ethernet. Ethernet and TCP/IP include the protocols used to pass files or data packets between nodes. FTP (File Transfer Protocol) is a specific means of file transfer that can be used under both Unix and OS-9 operating systems to transfer files between NPS AUV processes and the graphics simulation workstation.

Individual nodes connected to Ethernet have unique physical addresses. Because the NPS local area networks are also connected to the Internet, added nodes receive independent Internet addresses. The Internet is a national network connecting thousands of government, corporate and university computer networks. Internet includes links to all other major national and international networks. Utilizing off-the-shelf Ethernet technology for the NPS AUV Integrated Simulator is desirable since it allows complete compatibility with existing NPS networks as well as accessibility via the Internet.

Bandwidth is a critical consideration when choosing a network protocol. Large amounts of data are required for NPS AUV Integrated Simulator playback of vehicle performance, and numerous message outputs from many software processes may be required to adequately evaluate AUV mission execution. The network and protocol chosen must be capable of transmitting data at a rate adequate to support real-time playback and simulation. The data transfer mechanisms defined in Chapters III and IV contribute to efficient communication by concisely representing comprehensive vehicle state information. Current experience with interactive simulation programs (Ong 90) (Jurewicz 90) have shown that, under normal network usage rates, the Ethernet 10 M bps channel capacity is adequate to meet NPS AUV Integrated Simulator data transfer bandwidth requirements. Ethernet is therefore a good choice for the NPS AUV Integrated Simulator Data Network since most network connections were originally in place and network bandwidth is adequate to support simulation requirements.

The NPS AUV project may eventually need a special long distance connection to a proposed remote AUV laboratory in Building 230 at the NPS Golf Course. Building 230 is approximately 1.3 miles from Spanagel Hall, a distance that falls within the nominal 1.7 mile maximum range for an Ethernet network. However stringing Ethernet cable and repeaters along public roads to the golf course is obviously not feasible. Several options are available to maintain NPS AUV Data Network connectivity with a remote site such as Building 230. A leased high-speed phone line and additional interface equipment might be used to exchange real-time lab data with the Campus-Wide Network. In support of long-term research the Building 230 laboratory can be made an independent Internet node that includes both the NPS AUV and a local graphics workstation. Finally, a standard phone line and a modem connection can be used for file transfer to the Computer Science Department network as is already possible during pool testing. It is interesting to consider that standard telephone connections can also be used to communicate with the NPS AUV at other remote locations such as the Monterey Bay Aquarium, Navy laboratories or a coastal launch site.

Other network configurations and data link methods were also considered but deemed inappropriate. Direct customized serial or parallel port interfaces between an Iris graphics workstation and a laboratory AUV Gespac are possible but implementing such ports is expensive, time-consuming and incompatible with other networked computer systems at NPS. Such a customized communication setup also risks becoming obsolete if any of the current vehicle hardware is changed or upgraded. Direct port-to-port communication links also prevent linking multiple independent software processes running on an AUV microprocessor and the computer science network. Non-Ethernet network topologies such as token-ring or Fiber Distributed Data Interface (FDDI) were rejected for similar reasons.

A direct Ethernet connection inside the actual NPS AUV is operationally undesirable due to space and power consumption requirements. However addition of an optional Ethernet card to be used when the NPS AUV is in the laboratory provides

a high data transfer channel bandwidth (10 M bps) and improved pseudo-mission testing response. Actual AUV system reconfiguration to include the Ethernet card is not a problem since that is the normal configuration of the standalone laboratory AUV microprocessor.

C. NETWORK HARDWARE REQUIREMENTS

Numerous components are connected to form the NPS AUV Integrated Simulator data network, shown in Figure 5.1. Most network connections were already available at the start of this thesis. Notable construction work performed included assembly of the laboratory AUV Gespac and support equipment (node *auvsim1*) as well as the support laptop (node *auvsim2*) and the support PC (unconnected node *auvsim3*). The pertinent portions of the NPS Computer Science Department network are shown in Figures 5.2 and 5.3, and the NPS Campus-Wide Network is shown in Figure 5.4.

The most challenging task in the development of the NPS AUV Integrated Simulator data network was to build a laboratory AUV. An additional Gespac microprocessor identical to the NPS AUV Gespac microprocessor was purchased for simulation use. A VME-bus backplane cage containing a Gespac microprocessor, input/output channels and an Ethernet interface card was installed inside a cannibalized workstation box. A second backplane cage was combined with the first to provide extra capacity for evaluating additional cards prior to installation on the actual NPS AUV. A pair of monitor and keyboard hookups were connected to communicate with the microprocessor. The laboratory AUV was installed adjacent to the graphics simulation workstations, since the laboratory Gespac microprocessor and support terminals need to be directly visible from the simulation workstations for efficient testing and troubleshooting. This also made the lab AUV accessible to the local area network ethernet cable. The support PC includes an OS-9 "C" language cross-compiler which produces GESPAC 68020/68030 object code.

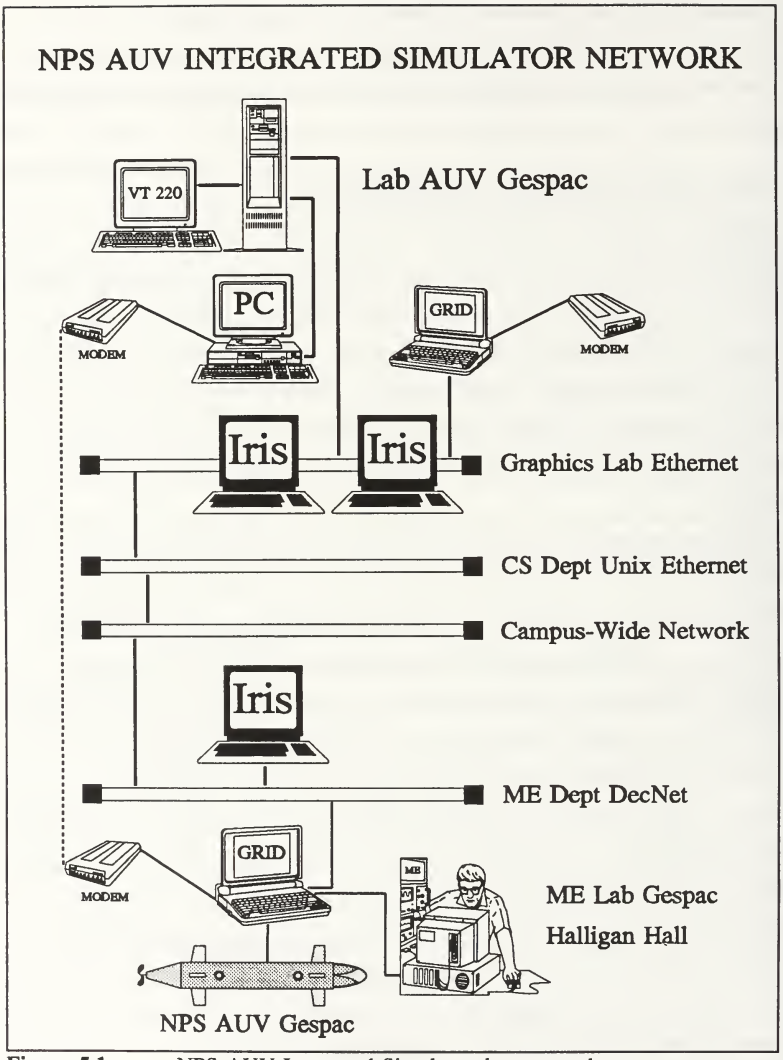


Figure 5.1 NPS AUV Integrated Simulator data network

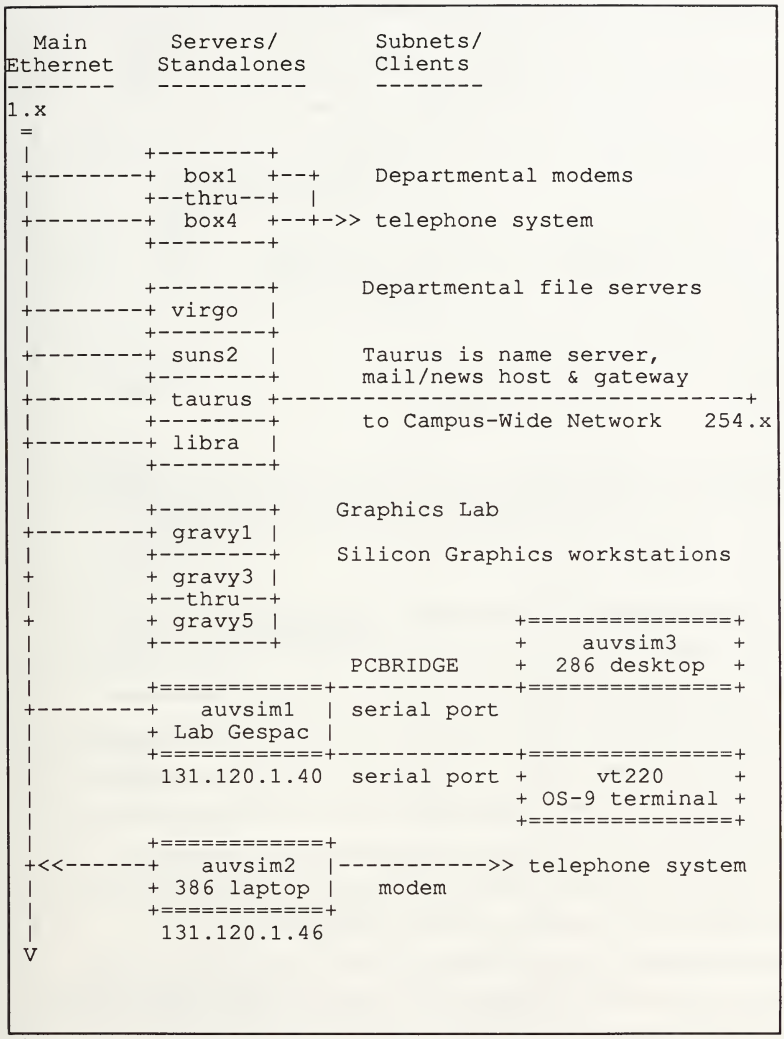


Figure 5.2 NPS Computer Science Department Network portion of NPS AUV Integrated Simulator data network (part 1)

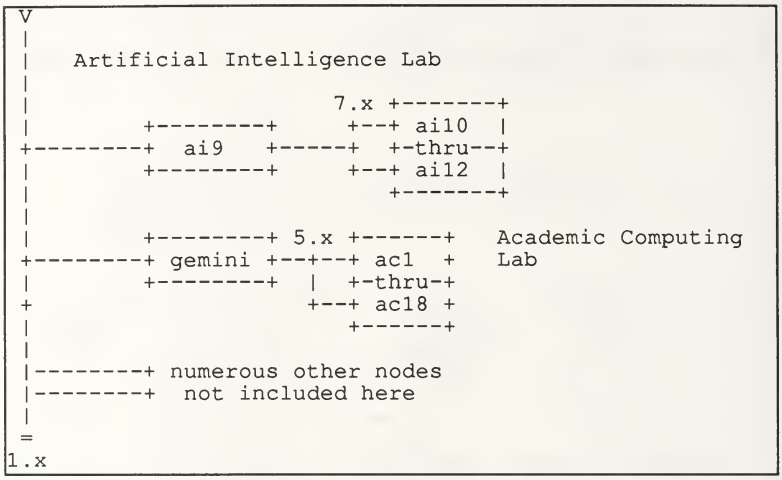


Figure 5.3 NPS Computer Science Department Network portion of NPS AUV Integrated Simulator data network (part 2)

The laboratory Gespac microprocessor is configured almost identically to the NPS AUV Gespac in order to allow realistic and thorough simulation. The laboratory Gespac differs from the actual NPS AUV Gespac by having an Ethernet card connection to the Computer Science Department network and serial port connections to a VT-220 terminal and a support PC. Care has been taken to preserve physical and functional equivalence between laboratory and NPS AUV Gespac microprocessors. The current laboratory AUV card cage configurations is shown in Figure 5.5.

The Gespac 68020/68030 microprocessor running OS-9 requires Erasable Programmable Read-Only Memory (EPROM) modifications for permanent configuration changes. However any desired operation can be performed without performing tedious a EPROM change because system drivers can be added as needed from the OS-9 command line. The current network configuration of the laboratory Gespac microprocessor will allow development and implementation of communications sockets to pass data packets for real-time simulator display.

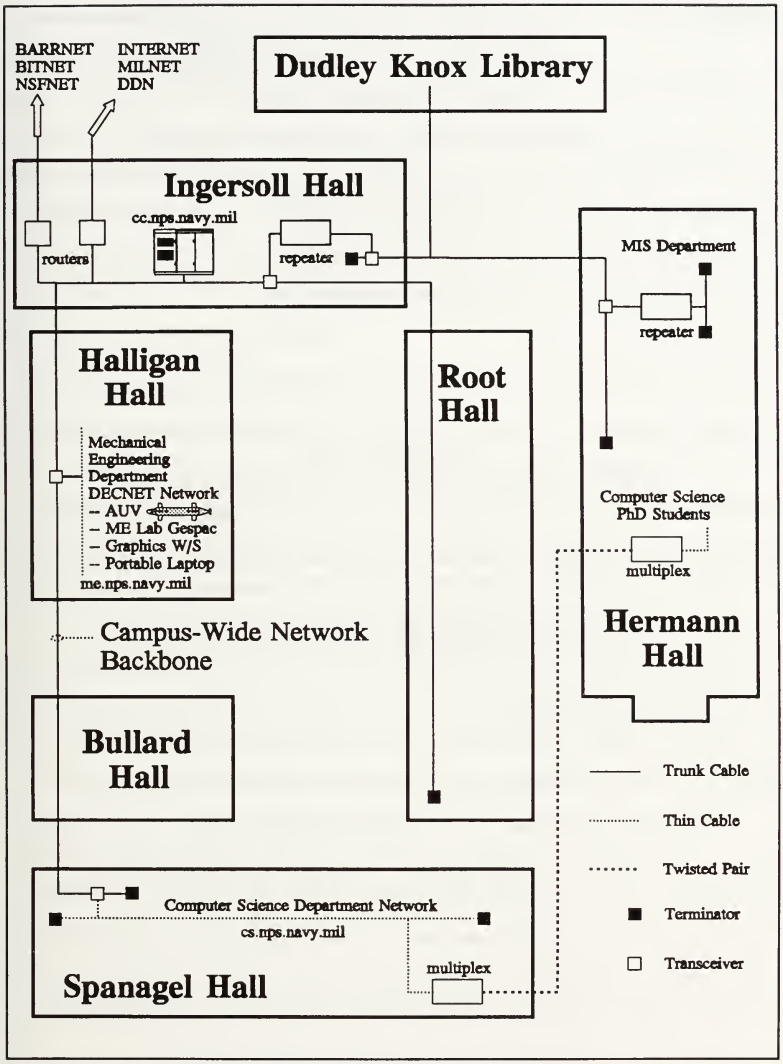


Figure 5.4 NPS Campus-Wide Network

1	**Gespac 68020 or 68030 Microprocessor** running OS-9 real-time multitasking operating system
2	**Multifunction Interface Card** 2 serial ports, 2 parallel ports, clock/calendar, etc.
3	**Floppy Disk Controller**
4	**Hard Disk Controller**
5	**Ethernet Interface Card**
6	
7	
8	**386 Microprocessor** with IDE controller circuitry (OS-9 compatibility untested)

1	Additional card slots available for future growth,
2	new equipment evaluation and hardware troubleshooting
3	
4	
5	
6	
7	
8	

Figure 5.5 Laboratory AUV microprocessor card cage slots

Prior to implementation of the NPS AUV Integrated Simulator data network, the Mechanical Engineering Department network was not connected to any external network. A special interface board was purchased and installed during 1991 that linked their DECNET-based system to the Campus-Wide Network backbone cable running through Halligan Hall. Another Iris graphics workstation is also on this local area network, permitting local execution of graphic simulation as part of the overall NPS AUV Integrated Simulator. Another laboratory Gespac is also available in Halligan Hall. Each of these nodes can now be networked to take advantage of the many benefits of integrated simulation.

D. OPERATING SYSTEM INTERFACES

The Computer Science Department network uses the Unix operating system. Although different versions of Unix can coexist on subnetworks connected to the departmental network, each is compatibly connected using Ethernet and Internet TCP/IP protocols. Further capability can be provided by socket interface software that allows application programs to directly access communication protocols via the operating system. Sockets for interprocess and intermachine communication have been written and implemented under Unix (Barrow 88) (Ong 90). The Gespac OS-9 operating system also supports Ethernet and TCP/IP protocols but further work is needed to implement software sockets under the Gespac OS-9 operating system.

Complex simulator testing of the NPS AUV may someday be based on real-time data transfer between AUV software processes and the integrated simulator. NPS AUV software processes have two possible operating environments: simulated developmental testing under the Unix operating system or operational execution under the OS-9 operating system on a Gespac microprocessor. Although file transfer is the only data passing mechanism currently implemented, it is desirable to develop real-time testing capabilities in both operating system environments by implementing sockets to pass data packets. If data transfer between processes succeeds independently of the NPS AUV under the developmental Unix environment, it is

reasonable to expect that the same software processes will still communicate properly when ported to the NPS AUV's resident OS-9 operating system. Careful implementation of data packet passing and network sockets will ensure proper portability of processes that employ sockets.

Use of ASCII text files for mission logging makes encryption a simple matter if increased security becomes necessary due to mission requirements. Encryption can be performed prior to file transfer or during high-level object file message logging. Any encryption routines used need to comply with the DoD Encryption Standard (DES) or higher requirement. Encrypted output files need to remain in ASCII text format for complete communications protocol compatibility.

E. CONCLUSIONS

Creating an NPS AUV Integrated Simulator Data Network by linking NPS AUV hardware with graphics workstations will provide an exceptional tool for distributed research, real-time simulation and end-to-end AUV system testing. Ethernet is the appropriate local area network technology to use in order to ensure complete compatibility with existing NPS networks. Ethernet and Internet connectivity ensures that the NPS AUV Integrated Simulator data network completely supports all NPS AUV research, development and testing. Further work remains to implement TCP/IP protocol and software socket compatibility under the OS-9 operating system running on the Gespac 68030 architecture.

Connection of the NPS AUV Integrated Simulator data network allows preliminary evaluation of NPS AUV software changes to be performed prior to in-water testing. Concurrent playback and data verification is now possible during pool testing by transferring telemetry replay files over telephone lines during NPS AUV test runs. Immediate problem diagnosis allows immediate correction and repetition of test runs that result in corrupted, faulty or unusual data.

Additional work needs to be done on the NPS AUV Integrated Simulator data network when modeling or simulating missing NPS AUV hardware in the graphics

lab. It is likely that some world models will not fit on the operational Gespac requiring data packet communication with an offline world model. The laboratory AUV is an excellent testbed for evaluating new potential NPS AUV physical components. Finally, the data network has the growth potential to support the addition of analog NPS AUV hardware components, allowing the NPS AUV Integrated Simulator to become a true digital and analog hybrid simulator.

VI. AUTONOMOUS SONAR CLASSIFICATION USING EXPERT SYSTEMS

A. ABSTRACT

An expert system can process active sonar returns, perform geometric analysis and autonomously classify detected underwater objects. Autonomous classification of objects is an essential requirement for independent operation by autonomous underwater vehicles (AUVs). Most AUVs are only capable of rudimentary sensor analysis, since standard approaches to evaluation and classification of sonar data require excessive signal processing and computational power to be practical. This chapter describes how to develop an autonomous sonar classification expert system for a working AUV.

A fundamental approach is presented for applying geometric reasoning and expert system heuristics to sonar classification. Preliminary sonar processing is performed using parametric regression line fitting. A polyhedron-building algorithm correlates the parametric regression line segments into geometric objects. After quantifying geometric object attributes, objects are classified using rule-based evaluation of quantitative and qualitative attributes combined with sonar classification heuristics.

A summary of expert systems describes their salient features pertinent to autonomous sonar classification systems. The expert system paradigm, knowledge representation, reasoning using facts and rules, rule sets, control of execution flow and expert system development are outlined. Expert system self-diagnosis and self-correction also discussed.

Implementation was performed using the "C" Language Integrated Production System (CLIPS) expert system shell. Real-time graphic simulation and scientific visualization are employed to evaluate results. Experimental sonar classification results are presented using actual mission data from the Naval Postgraduate School

(NPS) AUV. Successful classifications of walls and a mine-like object are demonstrated.

B. INTRODUCTION

Intelligent vehicles will play a major role in future underwater missions. A critical requirement for independent behavior by such vehicles is autonomous analysis of complex and variable ocean environments. This is a notoriously difficult task, even when human operators use sophisticated sensors and powerful processors.

Although much work has been done in vision processing for mobile robots, additional research has been needed on interpretation of observed scenes and terrain (Hebert 88). Numerous approaches to the general object-recognition problem are presented in (Besl 85). Both of these references can be found in (Iyengar 91), an essential collection of surveys, tutorials and fundamental research papers regarding mobile robot sensor perception, mapping and navigation. Other references included in (Iyengar 91) are (Luo 89) and (Moravec 83).

Independent and meaningful interpretation of sensor data is a principal prerequisite for accomplishing high-level AUV missions and behaviors. A number of universities and laboratories are conducting autonomous underwater vehicle (AUV) research and development that involves a wide variety of sensor types and sensor interpretation methods. The Defense Advanced Research Projects Agency (DARPA) Unmanned Undersea Vehicle (UUV) uses sidescan sonar and neural network classification for underwater mine detection (Pappas 91). Woods Hole Oceanographic Institution has used sidescan sonar, stochastic backprojection and a variety of vision processing techniques and sea floor shape information to create three-dimensional bottom images (Stewart 89). The University of New Hampshire Experimental Autonomous Vehicle (EAVE) III uses depth profiling, acoustic long baseline navigation and comparison with a world model to detect bottom objects (Blidberg 90). Numerous other examples of sensor data interpretation exist. In contrast to most methods, this sonar classification system uses parametric regression, geometric analysis

and expert system heuristics to create classifiable object types. An advantage of this method is that progressively higher levels of object abstraction are possible.

C. OVERVIEW

The objective of this chapter is to present a method for autonomous classification of underwater objects. This is achieved using geometric sonar analysis techniques and an expert system for heuristic reasoning.

This research effectively demonstrates that geometric analysis can be combined with an expert system to process, analyze and classify active sonar range and bearing data in support of AUV operations. Figure 6.1 shows how low-level sonar data is processed to produce increasingly complex geometric objects and high-level classification outputs.

Geometric analysis can distill large amounts of sonar data into useful information that can be used to make logical and informed decisions. The primary difficulty in geometric sonar analysis is that active sonar signal returns are inherently noisy and unconnected. Parametric regression is a robust method of least-squares line fitting that permits precise geometric analysis of range and bearing data (Floyd 91). Generated regression lines are provided to a polyhedron-building algorithm to create geometric objects. Geometric object attributes can then be compared to known object types through the rule-based pattern-matching capabilities of an expert system, resulting in object classification.

The possible types of object classes to be detected are typically limited in number and somewhat predictable given *a priori* knowledge of the underwater environment. Geographic objects to be detected and classified include the ocean bottom, sea mountains, valleys, rock outcroppings and walls. Biological objects include fish, kelp, scuba divers and large animals such as dolphin or whales. Man-made objects include ships, submarines, torpedoes, mines, nets, pipes and cables. These object classes of interest are listed in Table I. The relatively small number of underwater objects of interest simplifies sonar classification criteria. Primary expert

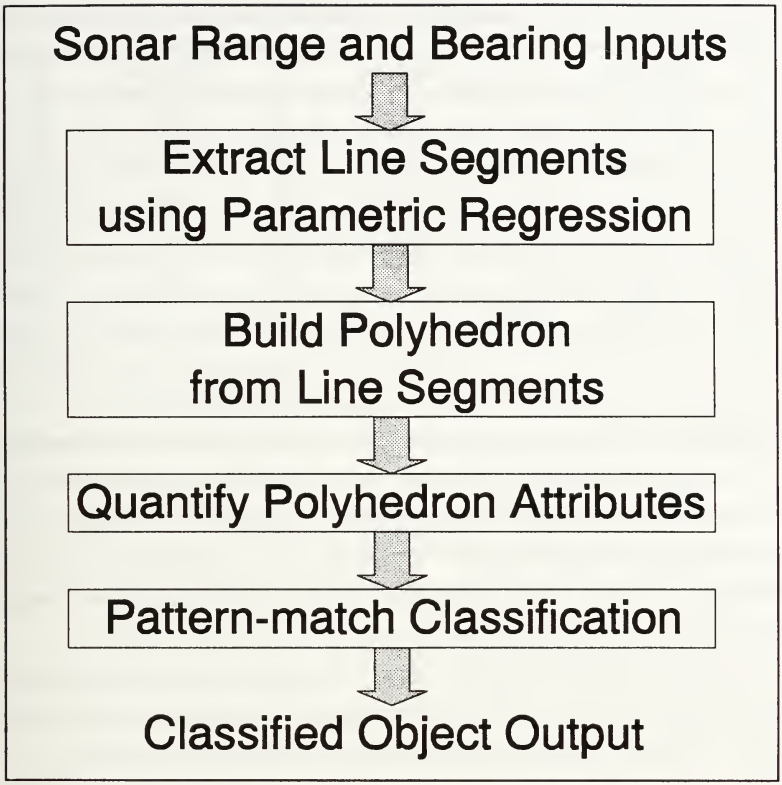


Figure 6.1 Autonomous sonar classification process diagram

system outputs are location, size, and classification of all sonar contacts.

Expert systems are an established methodology that can effectively and clearly represent specialized human knowledge using algorithms and heuristic rules. Typically the functions performed by an expert system might otherwise require human action by a knowledgeable expert. The expert system approach is applicable to a wide variety of complex problems, even when no single expert understands all aspects of a particular problem domain.

Table VI.1 EXAMPLE UNDERWATER OBJECT CLASSIFICATION TYPES

Geographic	Biological	Manmade
Ocean bottom	Fish	Ship
Sea mountain	Kelp	Submarine
Valley	Scuba diver	Mine
Rock outcropping	Dolphin	Torpedo
Wall	Whale	Net
Sea surface	Shark	Pipe or cable
Unknown		

The use of real world data is important for development and verification of a sonar classification expert system. Naval Postgraduate School (NPS) students and faculty have designed and built a working AUV that can be used to provide a variety of classifiable sonar data. Successful examples of expert system classifications using NPS AUV sonar data are described in detail.

The expert system approach also appears to be usable for sensor fusion using a wide variety of sonar types as well as non-acoustic sensors such as laser rangefinders and video. Many exciting future applications are possible using expert system methods.

D. GEOMETRIC ANALYSIS OF SONAR DATA

1. General Characteristics of Active Sonar Data

Outputs common to practically all active sonars are range and bearing from the sonar transducer to a contact, if any is detected. Posture of an underwater vehicle includes a three-dimensional position coordinate, as well as vehicle attitude consisting of roll, elevation and azimuth orientations. The relative position of each sonar return is combined with vehicle posture using vector addition to yield a precise

three-dimensional coordinate. In this chapter the term "sonar data" refers to simultaneous sonar range and bearing data returned from an active sonar transmission.

2. Geometric Primitives and Object Attribute Definitions

Sonar data can be analyzed to produce geometric forms such as points, lines or polyhedra. Precise definitions of geometric primitives and object attributes are necessary for predictable and repeatable sonar classifier performance. It is important that the theoretical basis of a sonar classification expert system be both mathematically rigorous and as general as possible in order to allow increasingly sophisticated analysis of data. A formal geometry-based approach also permits expert system compatibility with a wide variety of sonar types.

The geometric primitives considered by this expert system are point, line segment, polyhedron and cylindrical polyhedron (i.e. a three-dimensional polyhedron that extends vertically up and down from a planar polygon perimeter). Object attributes include centroid position, depth, length, width, height, perimeter, cross-sectional area, thinness, and volume. Indirect attributes such as positional accuracy, confidence factor, inferred edges and hidden edges are also evaluated.

Additional geometric primitives and object attributes can be defined as necessary to utilize the more sophisticated data available from sector scanning, two-dimensional swath or three-dimensional multi-beam sonars. Similar approaches using curved shapes such as circles, ellipses or conics (Moravec 83) are also compatible.

3. Extracting Line Segments using Parametric Regression

Linear relationships described by sets of discrete data are typically found using standard linear regression analysis, commonly known as least-squares fit. This method is widely used but has a significant limitation in that regression calculations on (x,y) coordinate points parallel to the y-axis result in divide-by-zero singularities for slope and mathematically undefined regression results. Since typical unconstrained sonar data may lie along any three-dimensional orientation, a different method is

needed for autonomous fitting of best-approximation line segments to a series of discrete sonar returns.

The parametric regression method utilizes a polar coordinate derivation of linear regression analysis to provide a robust and accurate least-squares fit of line segments to sequences of data points. This method has been developed in detail and is particularly well suited for geometric analysis of real-world sonar data (Kanayama 89) (Kanayama 90) (Floyd, Kanayama, Magrino 91) (Floyd 91). Associated with each regression line segment is an elliptical thinness term that can be used as a metric for line segment accuracy and data variance. Figure 6.2 shows a typical parametric regression line segment fit to a set of sonar returns.

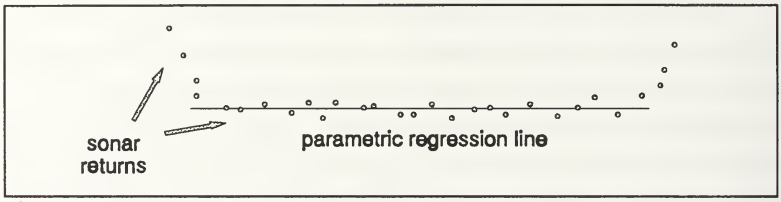


Figure 6.2 Typical parametric regression line fit

A further significant benefit of parametric regression analysis is that it is a sequential algorithm which provides immediate incremental improvements upon receipt of each individual data point. The sequential nature of this algorithm makes it highly suited for real-time operations that must meet immediate response requirements. Real-time vehicles cannot afford to wait for intermittently time-consuming sonar analysis when excessive delay might jeopardize navigational safety.

4. Building a Polyhedron from Line Segments

Parametric regression provides linear one-dimensional geometric primitives. However line segments by themselves are insufficient for thorough two-dimensional spatial reasoning or object classification. A polyhedron-building algorithm is presented here as a means of constructing two-dimensional geometric objects from a

sequence of regression line segments. In this context the polyhedron-building algorithm is a logical extension to the parametric regression algorithm.

One important assumption used when building polyhedra is that underwater contacts of interest have predominantly convex shapes, i.e. they contain no large concave depressions or cavities. This assumption permits clear delineation of independent object boundaries. Analysis of an actual concave object results in the definition of adjacent convex objects. Higher-level analysis at the heuristic level can be used to clump adjacent objects if needed.

Note that the orientation of vehicle sonar relative to detected objects is a critical consideration in the polyhedron building algorithm, since spatial relationships are equally dependent on sensor perspective and actual object shape.

Polyhedron building begins with a single line segment produced by parametric regression analysis of continuous sonar data. Each following segment from regression analysis on the same sensor is compared to the previous segment. If the follow-on segment meets proximity and orientation criteria, then it is considered to be another part of the same geometric object. This segment comparison process is repeated until proximity or orientation criteria fail, at which time the previous geometric object is complete and the follow-on segment becomes the beginning segment of a new geometric object.

Proximity is measured between the end point of the most recently correlated line segment and the start point of the next segment to be considered. The proximity criterion is typically small and restrictive (e.g. less than 1 foot) in order to permit discrimination between adjacent objects. The proximity criterion must be met prior to comparing relative orientation for geometric object extension.

Orientation comparisons are made to determine whether adjacent segments are colinear, convex or concave. The colinear test allows a reasonable error bound (e.g. $\pm 10°$) in order to account for sonar noise and line-fitting approximations. Colinear segments are acceptable for geometric object extension (Figure 6.3).

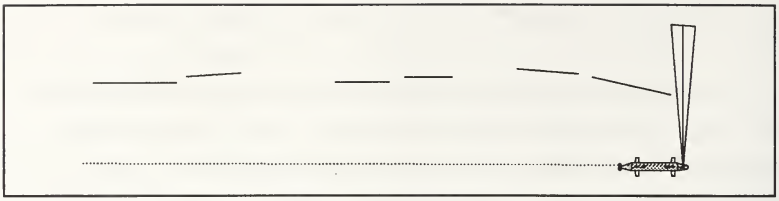


Figure 6.3 Examples of colinear regression line segments

The convex test measures whether the follow-on segment direction points farther away from the sensor's perspective than the previous segment. Convex segments are also acceptable for geometric object extension (Figure 6.4).

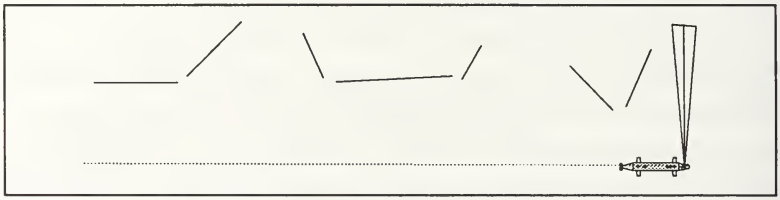


Figure 6.4 Examples of convex regression line segments

The concave test measures whether the follow-on segment direction points closer towards the sensor's perspective than the previous segment, in effect defining the boundaries of a hole. Concave line segment relative orientations indicate a break between separate convex geometric objects (Figure 6.5). The follow-on segment is used to start a new polyhedron.

Inferred edges are presumed to exist between each pair of the sequential detected edges that make up a polyhedron. A single hidden edge is presumed to exist between the start point and end point of a particular object. The classifier must recognize, however, that such hidden edges may be completely inaccurate since the actual hidden sides of the object were obscured from the sonar.

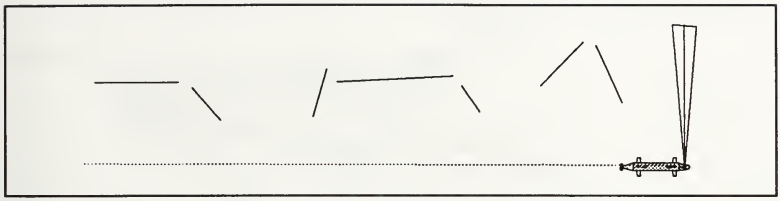


Figure 6.5 Examples of concave regression line segments

In summary, the polyhedron-building algorithm correlates regression line segments into two-dimensional polyhedral objects. This method enables the application of computational geometry techniques to analyze large volumes of discrete range and bearing data. Figure 6.6 illustrates the polyhedron-building algorithm.

5. Quantifying Polyhedron Attributes

The attributes that are used to classify objects need to be precisely defined and calculated, wherever possible. For example, attributes such as depth, length, width and height are directly measurable using calculated sonar positions. Object perimeter can be determined by first summing the lengths of all correlated line segments, and then adding the lengths of all inferred and hidden edges that are presumed to exist between detected edges. Figure 6.7 shows how the start point, regression line segments, inferred edges and hidden edge that make up a polyhedron cross-section define a series of triangular areas.

Area of a single triangle is given by Equation (6.1).

$$Area_{\Delta} = \frac{1}{2}\left[(X_2-X_1)(Y_3-Y_1) - (X_3-X_1)(Y_2-Y_1) \right] \qquad (6.1)$$

Polyhedron cross-sectional area is determined by summing the area of these triangles, given by Equation (6.2).

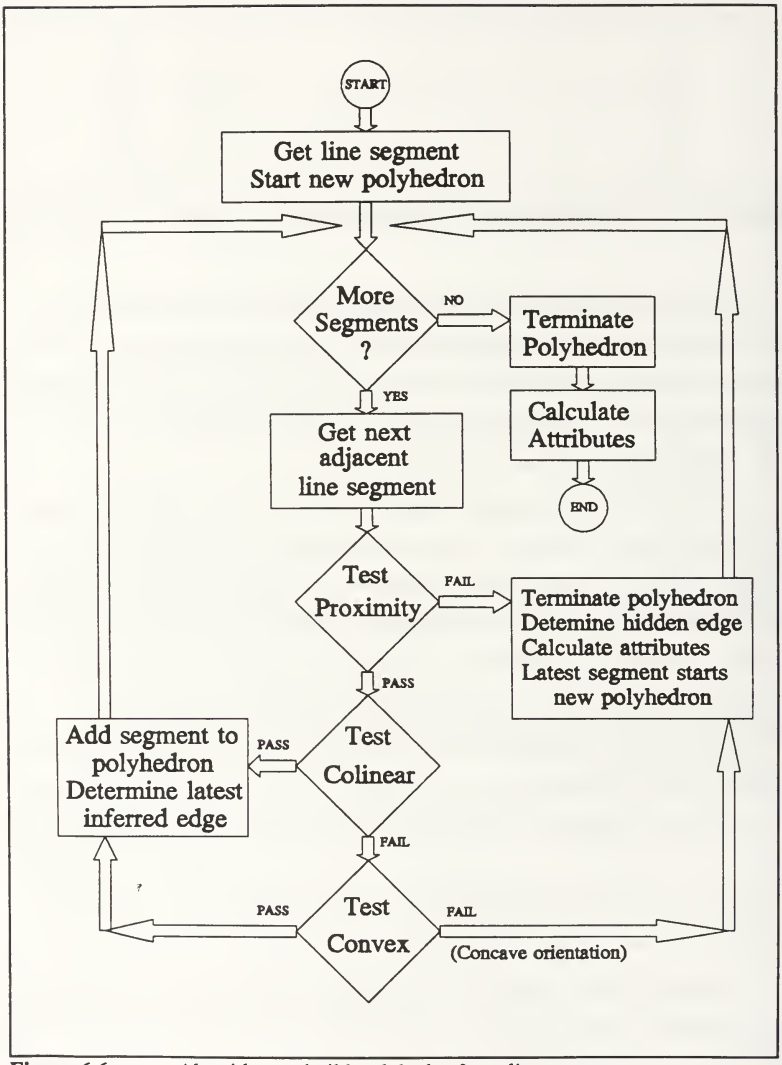


Figure 6.6 Algorithm to build polyhedra from line segments

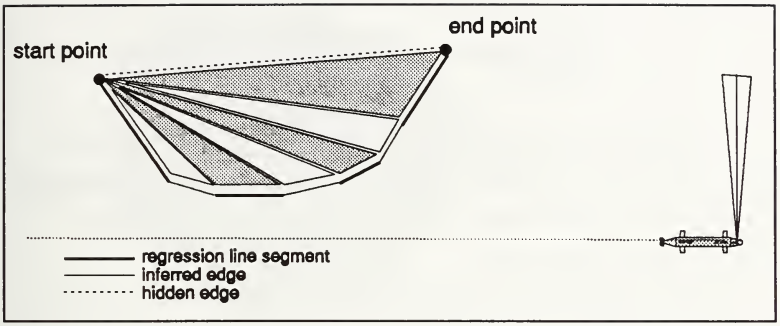


Figure 6.7 Summing triangle areas to determine polyhedron cross-sectional area

$$Area_{Polyhedron} = \sum_{\substack{regression \\ lines}} \left[Area_{\Delta\ start\ point,\ inferred\ edge} + Area_{\Delta\ start\ point,\ regression\ line} \right] \quad (6.2)$$

Centroid position for a triangle is calculated using Equation (6.3).

$$Triangle\ Centroid = (X_C , Y_C) = \left(\frac{X_1 + X_2 + X_3}{3} , \frac{Y_1 + Y_2 + Y_3}{3} \right) \quad (6.3)$$

Centroid position for the polyhedron cross-section is precisely determined by taking the weighted average of each of the triangle centroids, given by Equation (6.4).

$$Polyhedron\ Cross\text{-}section\ Centroid = \left(\frac{Area_{\Delta_1} X_{C_1} + \cdots + Area_{\Delta_N} X_{C_N}}{Area_{Polyhedron}} , \frac{Area_{\Delta_1} Y_{C_1} + \cdots + Area_{\Delta_N} Y_{C_N}}{Area_{Polyhedron}} \right) \quad (6.4)$$

Polyhedron cross-section thinness is defined as the ratio of polyhedron area to the square of polyhedron perimeter, given by Equation (6.5).

$$\frac{\textit{Polyhedron}}{\textit{Cross-section}} = \frac{\textit{Polyhedron Area}}{(\textit{Polyhedron Perimeter})^2} \qquad (6.5)$$
$$\textit{Thinness}$$

If object height is needed and has not been directly measured, it can be estimated using heuristic rules based on object depth, bottom depth or independent object classification. Object volume is the product of cross-sectional area and measured or estimated object height.

Indirect attributes such as positional accuracy, confidence factor, inferred edges and hidden edges are also evaluated. Point positional accuracy is derived by combining current vehicle positional accuracy estimate with sonar accuracy or sonar beamwidth at the range to the object. Confidence factor can be defined independently of positional accuracy as a measure of how well the object matches a classification rule. Hidden edge length is a measure of what is *not* known about the object. Defining initial classification confidence factor as the ratio between hidden edge length and detected perimeter further indicates how much of the contact has actually been evaluated. Hidden edge metrics can be used to indicate whether further sonar investigation of the contact is desirable. Figure 6.8 shows detected edges, inferred edges and hidden edge relative to processed sonar returns, and how these geometric primitives may not fully reveal all features of a contact.

E. EXPERT SYSTEM HEURISTICS FOR SONAR CLASSIFICATION

While geometric analysis can be defined with mathematical precision, human knowledge regarding sonar classification is less rigorous and can best be encoded as expert system heuristics.

1. Classification Heuristics and Attribute Heuristics

Sonar classification is not always a well defined problem. For example, it is possible that sonar analysis of a single object can be performed from different directions and lead to completely different classifications. An analogy to classifying objects using simple range and bearing sonars is attempting to identify your

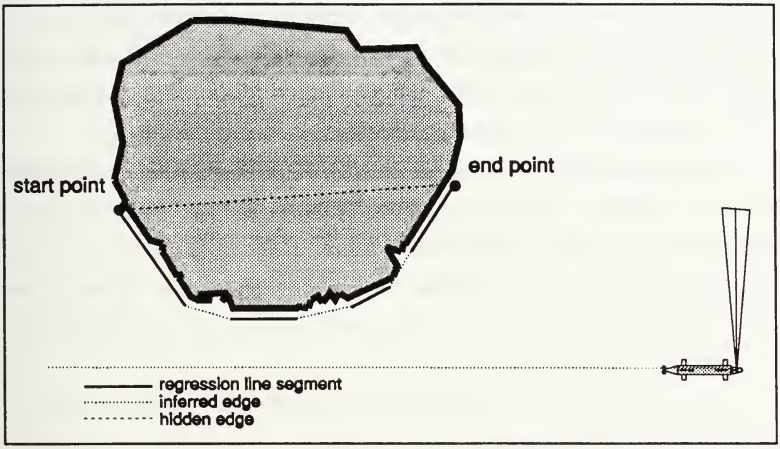


Figure 6.8 Polyhedron detected edges, inferred edges and hidden edge may not fully reveal all features of the sonar contact

surroundings while looking at the world through a steerable pinhole. It is difficult! Consequently, sonar classification criteria are often ambiguous and difficult to quantify, even when using formally derived geometric primitives. However, the heuristic approach used by expert systems is effective in many types of inexact problems and enables an autonomous system to obtain excellent sonar classification results.

Heuristics can be used for evaluating attributes such as object height when information is incomplete. Both attribute and classification heuristics can be easily modified in understandable ways despite the ambiguities of sonar analysis. The intuitive power of heuristics combined with the precision of geometric analysis gives sonar classification expert systems wide applicability and adaptability.

For this expert system approach, classification of sonar contacts is performed by comparing attributes of detected objects with predetermined attributes of known objects of interest. Different classification criteria are necessary and desirable for different environments. In particular, the different characteristics of deep ocean

versus shallow water versus an artificial pool will constrain the possible types of objects to be detected. Knowledge of the current environment can be extremely useful when determining the specialized classification rules and heuristic criteria to be used for a given mission.

Precise classification of every possible object type may not be necessary for some missions. Resolution of an ambiguous classification typically requires multiple sensor looks, costing additional time and energy. Preliminary classification as a potential contact of interest may be sufficient to justify maneuvering for additional sensing and closer investigation. Conversely, objects deemed to be of no interest require no further investigation by the vehicle.

Size can be the primary classification attribute for most underwater objects of interest. However, size per se is not a strictly defined term. It is worth mention that significant object size may be indicated by a variety of attributes including cross-sectional area, volume, perimeter, thinness or hidden edge length. Any or all of these size-related attributes may require close evaluation in order to properly discriminate between similarly sized sonar targets such as mines and rocks.

2. Pattern-match Classification Examples

Examples of how heuristic rules work can illustrate how a sonar expert system can classify objects. Two examples are presented here.

Preliminary wall classification is possible during the execution of the polyhedron-building algorithm. Walls are defined as any flat linear surface of non-trivial length. Polyhedra being built can be considered walls as long as each of the newly added regression line segments meet colinearity and proximity criteria. As soon as the polyhedron-building algorithm adds a new line segment based on convexity criteria, the polyhedron being built can be immediately reclassified from wall to object since the polyhedron is no longer linear.

Once a polyhedron has been built, all polyhedron attributes are automatically calculated. At this final stage, all of the preliminary work to quantitatively determine precise geometric objects greatly simplifies object

classification. For instance, a polyhedron might be classified as a mine-like object whenever cross-sectional area is between 10 and 100 square feet (Figure 6.9). Other objects can be classified in an equally straightforward manner.

Some objects should not be uniquely classified. For example, discrimination between a scuba diver and a mine-like object may be difficult. A particular strength of the expert system approach is that each object can receive multiple classifications with associated confidence factors as appropriate. This feature allows high-level reasoning using uncertainty, rather than being constrained by an arbitrary and potentially incorrect single classification.

```
(defrule classify-mine-like-object

; if this left-hand side of the rule is found to be true:

  ?poly <- (Polyhedron (status         COMPLETE)
                       (classification OBJECT)
                       (start          ?startpolytime)
                       (end            ?endpolytime)
                       (area           ?area))
=>

; then perform this right-hand side of the classification rule:

  (if   (and (>= ?area 10.0) (<= ?area 100.0)) ; area criteria test

   then (modify ?poly (classification MINE))

        (printout t "The polyhedron at times "  ?startpolytime
                                                ?endpolytime)
        (printout t "has classification MINE.")
)
```

Figure 6.9 Classification rule for a mine-like object

What was originally an intractable sonar classification problem is now much simpler and understandable at the highest level of the expert system.

3. Self-Diagnosis and Self-Correction

An additional strength of the expert system paradigm is that rules can be written to evaluate overall system performance, correcting internal vehicle problems without external control. Self-diagnosis is possible when expert system evaluation of sensor data differs from *a priori* knowledge of the real world. Such differences can be automatically fed back into the system to correct the offending error. As an example, gyro error and gyro drift rate can be diagnosed and quantified when a deduced wall orientation does not match known geographic data. Updating system estimates of gyro error and gyro drift rate result in an immediate improvement in sonar accuracy and positional estimates.

F. EXPERT SYSTEM PARADIGM

The power of an expert system is essential for a sonar classifier to perform high-level reasoning using qualitative attribute and sonar classification heuristics. This section describes the salient features of expert systems that are pertinent to the development of an autonomous sonar classifier.

1. Expert System Characteristics

An expert system typically includes the following characteristics: it simulates human reasoning about a problem domain, it uses symbolic knowledge representation and rules of thumb, it can analyze problems using heuristic or approximate methods that may not be guaranteed to succeed, and it deals with complexity that normally might require a human expert (Jackson 90). Expert system development differs from usual software engineering approaches in that rules of thumb can be developed incrementally to solve large problems that do not necessarily have a clearly defined solution methodology. Complementary rules work together without explicit supervision to discover solutions, if any exist.

2. Knowledge Representation and Reasoning using Facts, Rules and an Inference Engine

Expert systems typically use facts to represent knowledge about the state of the problem domain. Facts can be known prior to execution as part of the problem definition, and can also be discovered during program execution as new data becomes available or new knowledge is deduced. Rules are heuristic representations of human reasoning that follow the condition-action model. If a rule finds certain conditions to be true, then corresponding actions will follow and the rule is said to execute or "fire". The inference engine is the mechanism that allows all of the rules to individually examine the fact database and fire. The order of rule precedence and firing may range from random sequencing to a strictly defined execution order.

Strict execution order is typical of traditional programming paradigms but is usually considered to be an undesirable constraint for expert systems. Interestingly, random firing of expert system rules often uncovers solutions to problems that might otherwise be considered unsolvable using a strictly defined sequential approach.

3. Rule Sets and Control of Execution Flow

Given that a single rule may be inadequate to fully evaluate a complex situation, often groups of rules called "rule sets" are written to work together on particularly difficult analysis tasks. Such organization of rules allows a manageable and modular approach to expert system design. However, the random nature of rule firing allowed by an inference engine may permit partially processed facts to be accessed and used by other rules before the original rule set has completed the group objective. For this reason it is usually desirable to ensure that rule sets are able to run to completion whenever activated, before other rules are again allowed to fire. As an example, implementation of the algorithm in Figure 6.6 requires several polyhedron building rule sets working together in a coordinated fashion with parametric regression rule sets.

Given the unpredictable nature of heuristics when solving highly complex problems, the expert system designer may need to impose some controls on execution

flow among rule sets in order to ensure orderly execution. Randomness generally remains desirable and can still coexist within the bounds of rule execution flow control requirements.

4. Developing an Expert System

When a new application appears to be suitable for an expert system implementation, the first developmental step is to define the application problem in clearly understandable terms. This usually requires acquisition of expert knowledge in the problem area to be solved. The facts that may exist in the problem domain must be stated as unambiguously as possible. The overall problem can be logically grouped into simply stated subproblems consisting of condition-action rules (Jackson 90).

Once the problem is well-defined, facts and rules are converted from plain language into the syntax of the expert system being used. When first building an expert system, facts and rules should be added in small numbers. Incrementally test the expert system and avoid adding new rule sets until examples show that existing rules work as intended. Additional adjustment may be necessary to ensure mutual rule cooperation whenever new rules are added. Such an incremental prototyping approach can be particularly effective when building large expert systems (Sacerdoti 91).

G. IMPLEMENTATION AND EVALUATION

1. NPS AUV Vehicle Description and Sonar Characteristics

Naval officers and civilian scientists at NPS are conducting active research using an AUV designed and constructed at the school. The NPS AUV is used for basic research and thesis work in control systems technology, artificial intelligence, scientific visualization and systems integration. Specific NPS AUV project objectives include the study of mission planning, navigation, collision avoidance, real-time mission control, replanning, object recognition, vehicle dynamic motion control, and post-mission data analysis (Healey 91) (Brutzman Compton 91).

The NPS AUV is eight feet long and neutrally buoyant, displacing 387 pounds with overall size and shape comparable to a small dolphin. Current vehicle

endurance is two to three hours. Maximum speed of the NPS AUV is about two knots. The NPS AUV turning diameter is under three body lengths, designed to be ideal for maneuvering in the large NPS swimming pool. The NPS pool allows precise testing in a quiet, controlled environment. Open-ocean testing is feasible but is being reserved for a more robust follow-on vehicle. Video clips showing normal NPS AUV operation are available in (Brutzman, Floyd, Whalen 92) and (Brutzman 92).

The primary components of the NPS AUV are an aluminum hull, fiberglass sonar dome, four high-frequency directional sonar transducers, twin counter-rotating four-inch propellers, lead-acid batteries, eight plane surfaces, and a Gespac computer running a Motorola 68030 processor with a 2 MB RAM card. Figure 6.10 shows a general schematic of the NPS AUV.

Four PSA-900 Programmable Sonar Altimeters made by Datasonics Inc. are orthogonally fixed in the nose of the NPS AUV pointing directly ahead, downward and to port and starboard. These transducers are fixed frequency and ultrasonic, each at approximately 200 Khz. Sonar range gate is selectable at 30 m or 300 m, and pulse length is 350 μs. Normal pulse repetition rate is 10 Hz. Sonar beamwidth is seven degrees and range resolution is 1 cm at 30 m.

2. CLIPS Expert System

A number of expert systems are commercially available. CLIPS ("C" Language Integrated Production System) was chosen for this application due to its portability, extendability, capabilities, thorough documentation and interactive tutorials (NASA 91). CLIPS is also reasonably priced (approximately $450 or free for government agencies). CLIPS was developed by NASA to meet the varied requirements of NASA Mission Control Center delivery systems. CLIPS syntax is similar to the functional language Lisp and follows the if-then conditional rule model. The most recent versions of CLIPS add object-oriented and procedural programming capabilities (Giarratano 91). Since it is written in "C", CLIPS can run under most computer architectures. A feature of CLIPS that makes it particularly suitable for AUV use is that developed expert systems may be exported to nearly any

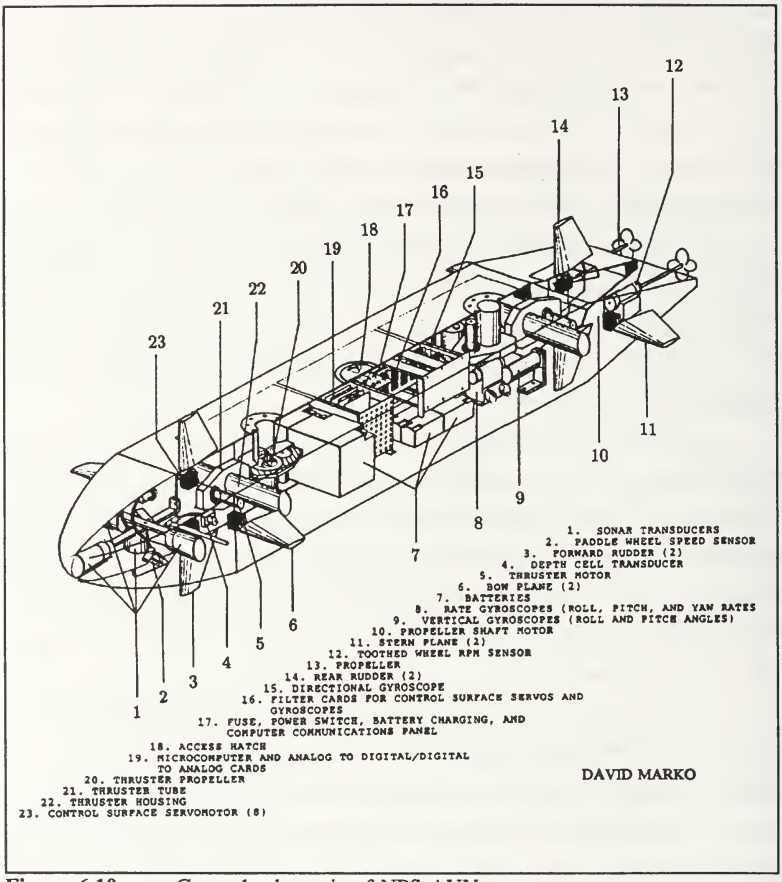


Figure 6.10 General schematic of NPS AUV

microprocessor by autogeneration of executable "C" language code. CLIPS has an active user base, annual applications conferences, an applications abstract registry and is provided with complete source code (Brooke 92).

3. NPS AUV Sonar Classification System

The program used to implement the concepts presented in this chapter was written using the CLIPS expert system. Actual sonar data collected by the NPS AUV is recorded in files for later use as input to the sonar classification expert system. This sonar data is analyzed off-line while running on a separate workstation.

A variety of outputs from the expert system provide several ways to visualize results. Two-dimensional graphics plots of raw sonar data and corresponding parametric regression line segments are shown on screen and as hard copy. An output file listing each individual geometric object and classification provides both hard copy of results and automatic input to the three-dimensional NPS AUV Integrated Simulator described below.

Sonar geometric analysis is computationally intensive. While running under the CLIPS environment on a Sun 2 workstation, the expert system is currently able to maintain a 7 Hz sonar return processing rate. This is nearly as fast as the 10 Hz data rate recorded by the NPS AUV and adequate for most real-time requirements. Optimization, elimination of network file server bottlenecks and source code compilation will further improve performance. Project goals include porting the NPS AUV sonar classification expert system to a microprocessor internal to the vehicle.

It is clear that a sonar classification expert system can operate autonomously in real time.

4. NPS AUV Integrated Simulator

Typically the development and testing of AUV hardware and software is greatly complicated by vehicle inaccessibility during operation. Integrated simulation remotely links vehicle components and support equipment with graphics simulation workstations. Integration of actual AUV components with three-dimensional simulation allows complete real-time, pre-mission, pseudo-mission and post-mission visualization and analysis in the lab.

Integrated simulator testing of AUVs is a broad and versatile method that has proven very effective in the development of the NPS AUV sonar classification expert system (Brutzman March 92) (Brutzman May 92). In particular, post-mission simulator playback of recorded telemetry, sonar sensor data and system state transitions supports in-depth reenactment, playback and analysis of processed sonar data. This scientific visualization approach permits rapid and precise development of geometric analysis techniques and classification heuristics for the NPS AUV sonar classification system.

High-resolution three-dimensional graphics workstations can provide real-time representations of vehicle dynamics, control system behavior, mission execution, sonar processing and object classification. Use of well-defined, user-readable mission log files as the data transfer mechanism allows consistent and repeatable simulation of all AUV operations.

H. EXPERIMENTAL RESULTS

1. Classification Test Scenario

An example best demonstrates successful classification of actual sonar returns. A single swimmer was chosen to represent a mine-like object and was positioned as a target near the right-hand wall of the NPS swimming pool, shown in Figure 6.11.

The NPS AUV was programmed to follow a racetrack traversal of the pool and record all pertinent data. Figure 6.12 shows individual left transducer sonar returns plotted as circles and vehicle track as a large oval, while the line segments calculated by the parametric regression algorithm are shown superimposed. Some distortion is evident due to unmodeled sideslip error in vehicle track data.

2. Experimental Results

The sonar data recorded by the NPS AUV in the pool are uploaded after mission completion via modem and processed off-line by the authors' sonar classification expert system. Classification results are then graphically rendered by the

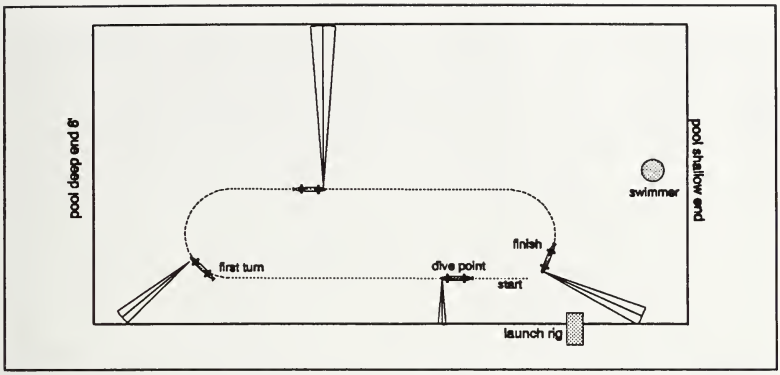


Figure 6.11 NPS AUV test track using left transducer only. Note swimmer target.

NPS AUV Integrated Simulator running on a Silicon Graphics Iris workstation. This three-dimensional display shows all generated parametric regression line segments, inferred edges, hidden edges, and detected walls. The overall pool graphics display as seen from a viewpoint high above the pool is shown in Figure 6.13. The target of interest met classification criteria for a mine-like object and a simulation closeup is shown in Figure 6.14.

The integrated simulator has the additional feature of being able to play back sonar detections and classifications simultaneously with vehicle motion in real time or slow motion. Evaluation of sonar classification results using the scientific visualization techniques provided by the integrated simulator was extremely helpful during development and testing of sonar expert system classification heuristics.

The experimental results show that the NPS AUV Autonomous Sonar Classification System is highly effective at classifying objects despite the low resolution of the active sonar employed.

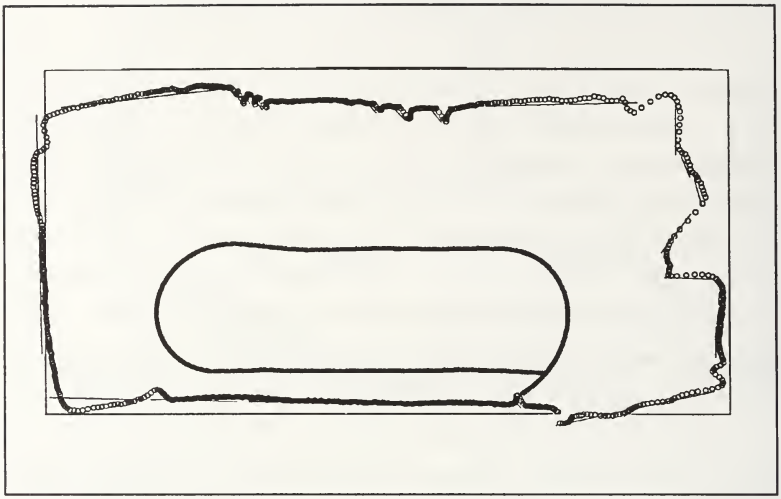


Figure 6.12 NPS AUV sonar classification expert system plot of pool data and parametric regression line segments

I. DISCUSSIONS AND APPLICATIONS

1. Extendability to Video, Lasers, Complex Sonars and Sensor Fusion

Active sonar is not the exclusive sensor used for underwater object detection and classification. A variety of other sensors are coming into use including underwater videocameras and lasers. In addition to range and bearing data, advanced sonars may provide completely different types of data such as frequency spectra, doppler or long-range conical beam data. Ultrasonic sonars have also been employed by land vehicles.

All of these sensors share common characteristics that allow autonomous analysis by expert systems. Each sensor type provides data sets that can be analyzed using geometric reasoning techniques. In every case expert knowledge can define both quantitative and heuristic rules for processing sensor outputs to create primitive geometric objects, thus allowing object classification.

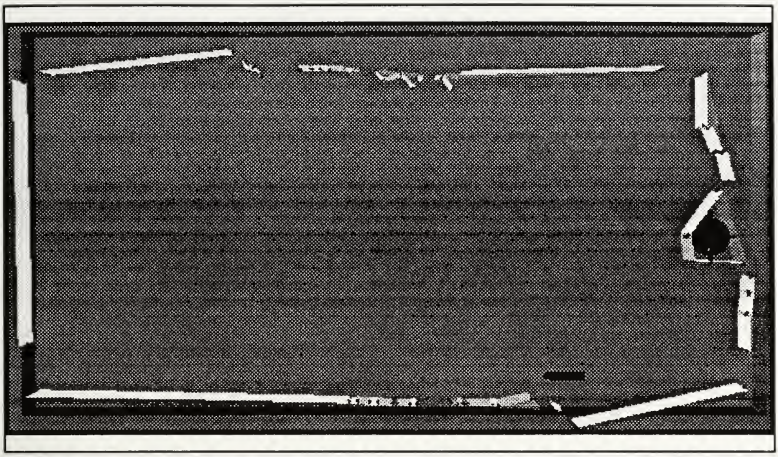


Figure 6.13 Integrated simulator screen display of the full NPS pool and all sonar classifications

Sensor fusion is the correlation of multisource information to resolve ambiguity and increase confidence in individual classifications. Sensor fusion is particularly valuable in offsetting the weaknesses of one sensor type with the strengths of another. An example of sensor fusion might be to correlate accurate laser bearing data with accurate sonar range data. A thorough survey on multisensor fusion roles, approaches and applications is provided by (Luo 89). Sensor fusion can be directly implemented using the pattern-matching capabilities of a multisensor classification expert system.

2. Intelligent Remote Sensors

The use of remote sensors is becoming commonplace. The primary limitation of most remote sensors is that they have little ability to independently react to sensor inputs. Most sensing devices require direct control or have an arbitrary sampling period, while continuously-sensing devices require dedicated data communication lines. Remote underwater sensors need to operate autonomously or

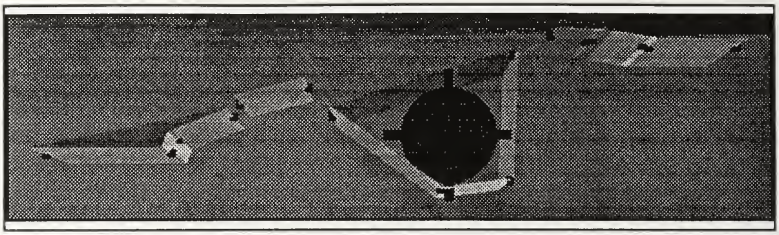


Figure 6.14 Integrated simulator display close-up of a mine-like object classified by the sonar expert system using detected edges, inferred edges, hidden edge and cross-sectional area

with minimal external control in order to improve their efficiency, capabilities and cost-effectiveness. Embedding an expert system application using microprocessor-based control is a feasible method to create intelligent and autonomous remote sensors.

3. Data Reduction

Most sensor data is high bandwidth. Autonomous vehicles, remotely operated vehicles and remote sensors typically receive extremely large amounts of data. Storage or transmission of raw data for off-line processing is undesirable and imposes unreasonable memory capacity and communications requirements. A significant benefit of autonomous classification is that it reduces massive amounts of raw data into concise information that can be efficiently recorded or communicated. Data without value is easily filtered. The overall data compression ratio can equal several orders of magnitude.

4. Future Use of Expert Systems by Autonomous Vehicles

If autonomous vehicle sensors and missions are to become increasingly capable and sophisticated, it is likely that parallel processing of distributed artificial intelligence modules will be necessary in order to provide adequate computing power with real-time response. A typical set of high-level processes might include detection and classification for multiple sensors, path planning, search, systems control and

others. None of these processes is completely independent, but typically each process can run in parallel with the others most of the time. One abstract software architecture that supports such a distributed approach is the blackboard paradigm.

A blackboard system directly extends the functionality seen in an expert system for a collection of distributed processes (Jackson 90). A good metaphor for the blackboard approach is a group of human experts working together on a large problem using a blackboard as their means of communication. Problem definition, data, questions and answers can all be written and read on various sections of the blackboard. Each independent expert has full access to the blackboard and looks for information pertinent to his area of expertise. When an expert develops some result or new question worth communicating to the group, that information is recorded on the blackboard.

Similarly, a blackboard system has distributed independent knowledge sources, each of which can use any method desired to solve portions of a large problem. Communications are recorded on the blackboard and are available to all knowledge sources. Complex problems are solved through cooperative reasoning (Corkill 91). As another example, each of the processes shown in Figure 6.1 might be implemented as separate knowledge sources for a blackboard. Expert systems are well suited as knowledge sources for the blackboard paradigm.

Development of autonomous expert systems is likely to provide intelligent components that will remain useful in the advanced architectures of future autonomous vehicles.

J. CONCLUSIONS

Autonomous sonar classification systems can accurately detect and classify objects in the underwater environment. Precise geometric analysis is combined with qualitative expert system heuristics to provide a flexible and robust approach with wide applicability. Autonomous classification systems are capable of supporting sophisticated real-time applications in working autonomous vehicles.

VII. SHORTEST PATH PLANNING USING A CIRCLE WORLD

A. ABSTRACT

Path planning is a critical capability for mobile robots operating in environments containing obstacles. Building a path planning module as part of this thesis has been helpful in understanding design requirements, world models and software architecture specifications both for the NPS AUV and the NPS AUV Integrated Simulator.

Mobile robot path planning around obstacles can be accomplished by modeling obstacles as pairs of identical circles with opposite rotations. Addition of robot radius and safety standoff distances to circle radii allows modeling the robot as a point. This circle world model can be used to calculate shortest paths between points.

Tangents between individual circles in a circle world have no inherent redundancy due to the geometric uniqueness of each landing and leaving point on every circle. Alternate partial paths landing at an intermediate circle obstacle must be properly compared in order to determine which is shortest. Dijkstra's algorithm or (preferably) A* search can selectively use visibility and partial path comparison calculations to find the shortest, safest or optimal path between start and goal points.

Tangent visibility from a single point to all circles can be calculated in order $O(n\log n)$ time. Similarly, tangent visibility from a single circle to all other circles can be calculated in order $O(n\log n)$ time. The shortest path between start and goal points can be calculated in order $O(n^2\log n)$ time.

Since obstacle avoidance is a typical robot behavior regardless of obstacle height, the circle world search model is directly extendable to the general case of three-dimensional path planning. This approach is shown to be particularly suitable for underwater vehicle path planning.

B. INTRODUCTION AND PROBLEM DESCRIPTION OF CIRCLE WORLD

Robot path planning is the search for an allowable, safe or optimal path for a robot to follow from one location to another. A critical consideration in robot path planning is the choice of model to represent the obstacles that a robot must avoid.

A well-known and fundamental method used for path planning is exemplified by configuration space approach (Lozano-Pérez 79) (Yap 87) (Akman 87) (Laumond 87) (Schwartz 88) (Canny 88). In the configuration space approach a world is modeled as a set of geometric obstacles. Obstacle boundaries are grown to include the effective radius of a mobile robot. The mobile robot center can then be treated as a reference point. Any location in the remaining non-obstacle free space is considered a legal position for the mobile robot reference point. Visible tangents can then be calculated between all obstacles. A fully connected graph is defined by obstacle boundaries and the tangents between them. Determination of a shortest path is accomplished by searching the visibility graph for the lowest cost path between start and goal points. Polygons are typically the geometric form chosen to represent obstacles.

Another simple and effective way to represent obstacles in a configuration space world model is to draw circles around them. The center coordinates of each circle in the circle world model are located at the centroid of each corresponding obstacle. The initial radius of each circle equals the minimum radius which completely surrounds the given obstacle (Figure 7.1). The maximum radius of the moving robot is combined with desired safety standoff distance and added to each obstacle circle radius (Figure 7.2).

Such a circle world model allows a mobile robot to be represented as a moving point. Any path in this circle world which can be drawn without crossing the interior of a circle boundary is a valid robot path.

As the number of obstacles in a circle world increases, the possible number of tangential paths from a start point to a goal point rises exponentially due to combinatorial explosion. A simple circle world that includes all visible circle tangents quickly becomes crowded (Figure 7.3). A typical circle world of moderate density has

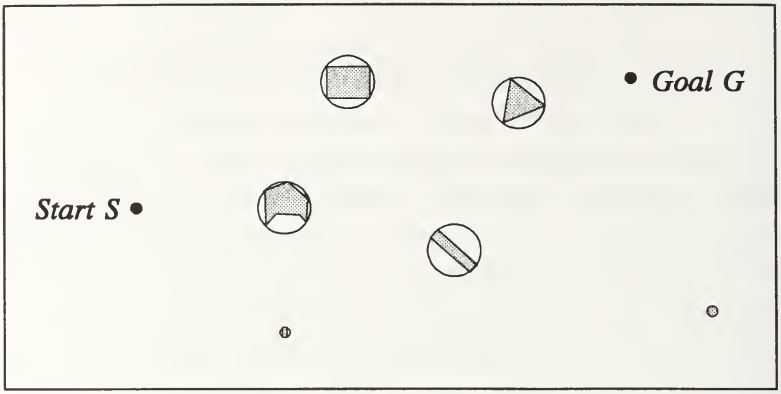


Figure 7.1 Simple obstacle representation using circles

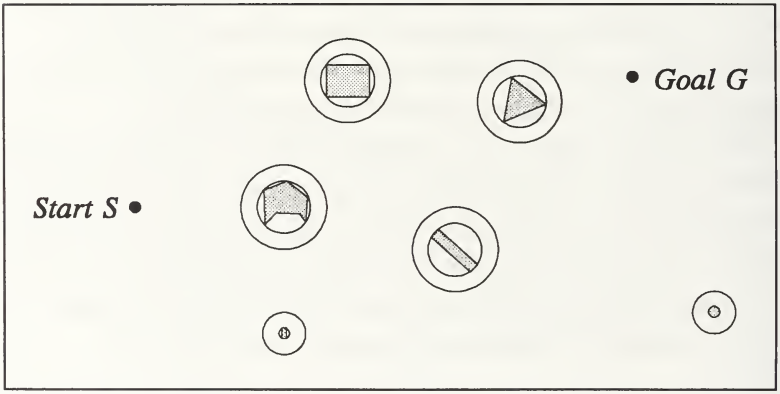


Figure 7.2 Improved obstacle representation including robot radius and safe standoff distance

about half of the circle tangents traversable while the remaining tangents are blocked.

The normal objective of path planning through an obstacle field is robot travel from a known start point to a known goal point. Typically a determination of shortest path is desired, and therefore Euclidean distance traveled is the metric used to determine the best route from start to goal (Figure 7.4). Other path selection criteria such as safety or optimality may also be considered (Kanayama 88). The circle world

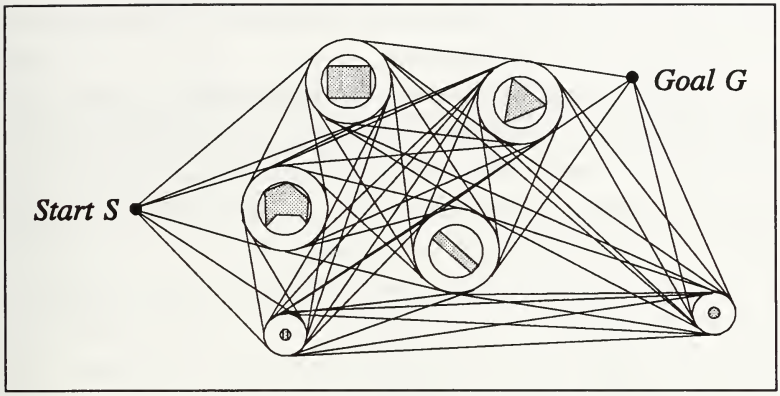


Figure 7.3 Simple circle world with all visible tangents

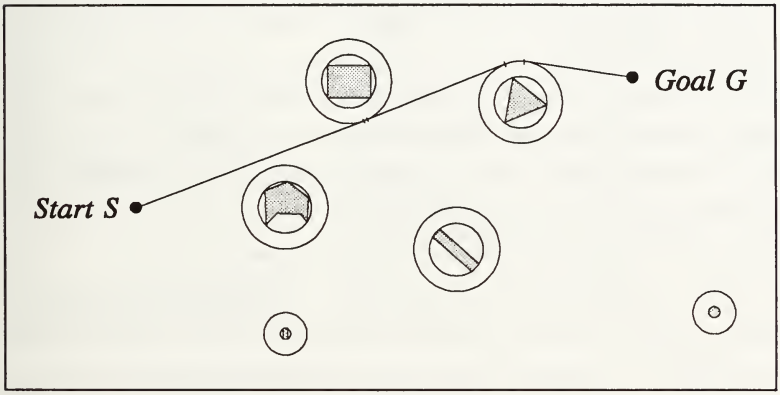


Figure 7.4 Simple circle world shortest path

model and path planning algorithm presented here allow rapid and efficient determination of shortest-distance paths.

At least one previous application, the Stanford cart mobile robot, used a similar circle world model for obstacle representation and avoidance (Moravec 80) (Moravec 83). However, the combination of vision processing and path planning

aboard that small robot proved prohibitively slow for real-time use due to hardware limitations and greater algorithmic complexity.

Although most configuration space treatments are based solely on linear or polygonal obstacles, obstacles composed of any closed combination of line segments and circular arcs are formally addressed in (Laumond 87). However that work does not explore the complexity differences between circle worlds and polygonal worlds for tangent computation and shortest-path search algorithms.

It is expected that the shortest-path planning algorithm provided in this paper will support real-time path planning by autonomous robots. These results are presented with additional mathematical theory and greater detail in (Kanayama Brutzman 91), which is included as Appendix D.

C. GEOMETRIC CHARACTERIZATIONS OF CIRCLE WORLD AND SHORTEST PATH

In a circle world, obstacles are modeled by surrounding them with circles. However, each circle surrounding an obstacle can have two possible traversal rotation modes: clockwise or counter-clockwise. In keeping with convention (Kanayama 91), clockwise traversals of circle perimeters can also be referred to as minus or left-handed, while counter-clockwise traversals can be referred to as plus or right-handed. A two-dimensional space filled with n noncontiguous obstacles can be fully represented by n clockwise circles and n corresponding counter-clockwise circles. Thus each obstacle is represented by two circles with opposite rotation modes, both centered at the coordinates of the obstacle centroid.

The geometric data primitives needed to fully represent a circle world are Point, Segment, Circle, Tangent, Arc, Configuration and Path. These geometric primitive data structures are defined here using a hierarchical approach for simplicity and clarity. When capitalized, these terms refer to the explicitly defined data structures summarized in Table VII.1. These data structure definitions are included in order to

best explain the circle world problem as well as the circle world source code in Appendix E.

Table VII.1 CIRCLE WORLD GEOMETRIC DATA STRUCTURES

Circle World Geometric Primitives Data Structures	Structure Elements	Element Data Types
Point	x, y	float
Segment	point1, point2	Point
Circle	center radius direction	Point float CW (-1), CCW (+1) or POINT (0)
Tangent	circle angle	Circle float
Arc	circle angle1, angle2	Circle float
Configuration	tangent orientation	Tangent float
Path (note: arc-segment pairs are repeated as necessary)	initial_segment ••• arc segment	Segment ••• Arc Segment

A Point in a circle world is defined by a set of two-dimensional Cartesian coordinates. A Segment is the line segment defined by two points that are connected by a straight line. A Circle data structure is defined by a Point, a radius and a direction of rotation. A Tangent data structure is comprised of a Circle and an angle which is the orientation between the circle center and the tangent point on the circle circumference. An Arc is defined by a Circle and two angles, corresponding to the starting and finishing angles oriented from the circle center to the arc starting and

finishing endpoints. Note that Arc direction of rotation is not dependent on the precedence of the starting and finishing angles, but rather is implicitly included in the Arc by the rotation direction of the member Circle. A Configuration is a combination of a Tangent and an orientation angle. The orientation angle of a Configuration is always perpendicular to the component Tangent angle but can be in either of two directions. Thus a Configuration point defines whether a Tangent is a landing point or a leaving point. Finally, a Path between a pair of start and goal points in a circle world includes a Segment followed by zero or more Arc-Segment pairs. A typical Path includes a straight line segment from the start point to the first circle obstacle, an Arc around a portion of that circle, a Segment from the first circle to the next circle and so on until the desired goal point is reached. As an example, a single segment Path could directly connect the start and goal points if no circle obstacles were between them.

A special case arises that is of use when designing visibility algorithms: a Point can also be treated as special case of a Circle. Setting a Circle radius value to zero or rotation mode to zero (or both) effectively makes that Circle behave as a Point. In particular this approach allows treating the start and goal points as circles, simplifying evaluation of tangents and paths in a circle world.

Every pair of circles defines four mutual tangent line segments: two external tangents and two cross-tangents (Figure 7.5). Note that the eight tangent points on the circumferences of these two circles are unique. Also note that the respective lengths of the external tangent segments and cross-tangent segments are equal.

The coordinates of the four possible tangents between the pairs of circles shown in Figure 7.6 can be calculated using Equations (7.1), (7.2) and (7.3). Rotation mode values follow the convention counter-clockwise (CCW, right-handed, plus or +1) and clockwise (CW, left-handed, minus or -1). Points which are being modeled as special cases of circles have rotation mode and radius equal to zero.

Angle δ in Equation (7.1) and Figure 7.6 is the offset angle between the circle-center-to-circle-center orientation θ and tangent orientation α.

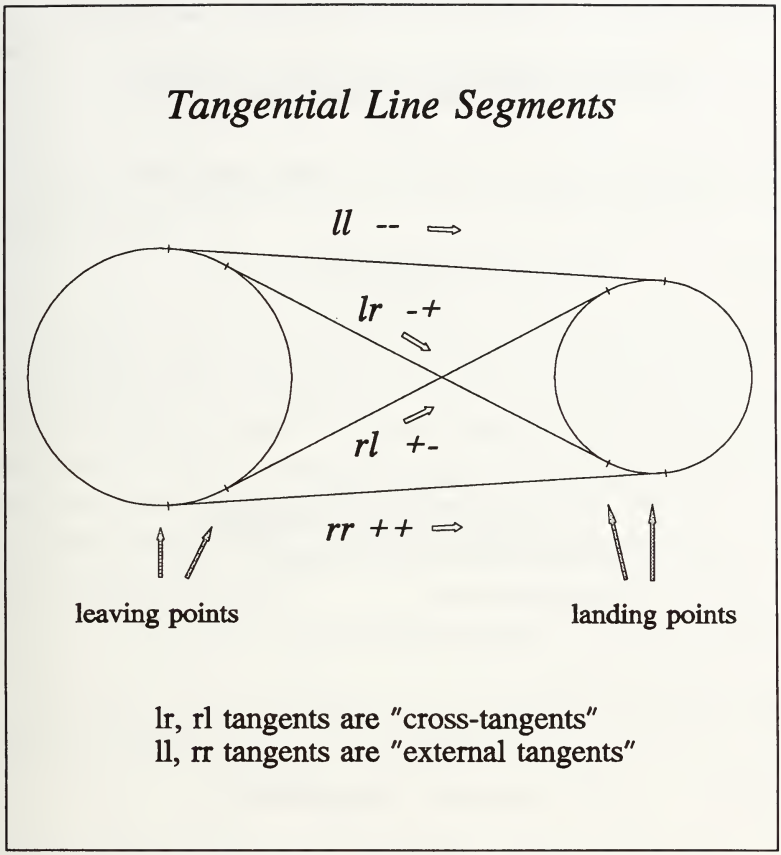


Figure 7.5 Tangential line segments between circles

$$\delta = \arcsin\frac{mode_2 \bullet r_2 \; - \; mode_1 \bullet r_1}{distance(circle_1.center,\; circle_2.center)} \qquad (7.1)$$

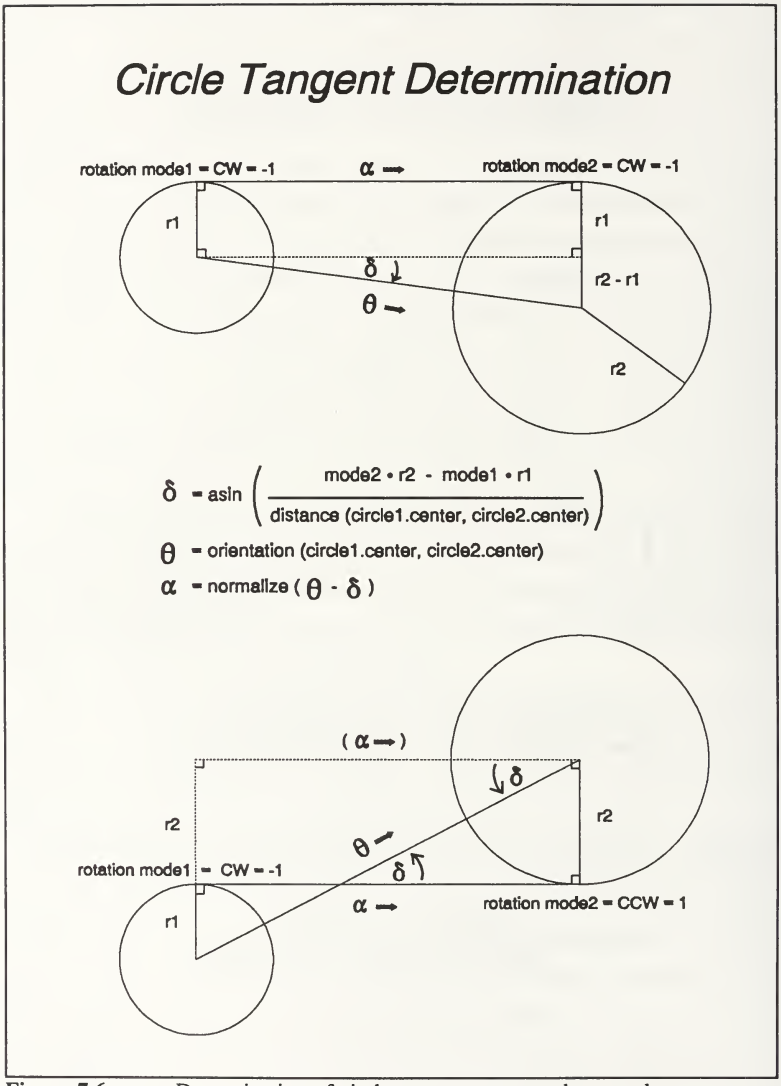


Figure 7.6 Determination of circle cross-tangents and external tangents

Angle θ in Equation (7.2) and Figure 7.6 is the orientation between circle centers.

$$\theta = orientation\ (circle_1.center,\ circle_2.center) \tag{7.2}$$

Angle α in Equation (7.3) and Figure 7.6 is the orientation of the desired tangent between the circles.

$$\alpha = normalize\ (\theta - \delta) \tag{7.3}$$

Visibility in circle world is defined as the ability to connect two points using a single line segment without crossing any circle boundary. The first step in determining visibility is to evaluate point-to-point segment orientation relative to every circle center using Equations (7.4), (7.5) and (7.6). The three regions of possible circle locations relative to the point-to-point segment are shown in Figure 7.7.

Angle θ in Equation (7.4) and Figure 7.7 is the orientation between the two points being checked for visibility.

$$\theta = orientation(point1,\ point2) \tag{7.4}$$

Angle $\delta 1$ in Equation (7.5) and Figure 7.7 is the angle between the current circle and line segment left side.

$$\delta_1 = orientation(point1,\ circle_i.center) - \theta \tag{7.5}$$

Angle $\delta 2$ in Equation (7.6) and Figure 7.7 is the angle between the current circle and line segment right side.

$$\delta_2 = orientation(point2,\ circle_i.center) - \theta \tag{7.6}$$

To complete the visibility check for two points and a given circle, the circle radius is now compared to the distance between the closest endpoint and the circle

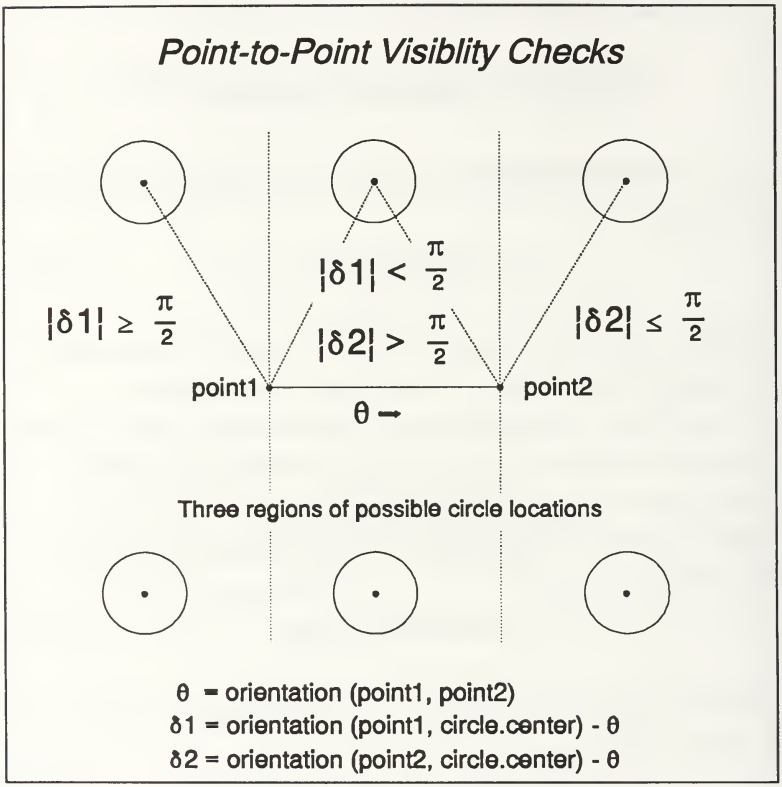


Figure 7.7 Determining point-to-point visibility in circle world

center. Three comparison results are possible. In the left-most region of Figure 7.7, successful test of Inequality (7.7) indicates that the current circle radius should be compared to the distance to point1 for visibility determination.

$$|\delta_1| \geq \frac{\pi}{2} \tag{7.7}$$

In the center region of Figure 7.7, successful test of both Inequalities (7.8) and (7.9) indicate that the circle radius in question should be compared to the distance to both point1 and point2 for visibility determination.

$$|\delta_1| < \frac{\pi}{2} \tag{7.8}$$

$$|\delta_2| > \frac{\pi}{2} \tag{7.9}$$

In the right-most region of Figure 7.7, successful test of Inequality (7.10) indicates that the current circle radius should be compared to the distance to point2 for visibility determination.

$$|\delta_2| \leq \frac{\pi}{2} \tag{7.10}$$

This segment visibility determination process must be repeated until any circle intersection disproves visibility, or until all circles in the circle world have been checked without intersection. A segment that touches only the perimeter of a circle without crossing the circle boundary can be considered visible. Visibility determination for a single pair of points in a world containing order $O(n)$ circles has algorithmic complexity of order $O(n)$.

The primary objective of path planning is to find a safest or shortest path. A simple measure of path costs in a circle world is the straightforward summation of Euclidean distances along segments and arcs. However, comparing relative costs when two partial paths have different landing points on the same circle is more complex. Although the first partial path can reach the intermediate circle obstacle using a shorter route than a second partial path, the second partial path can be part of a shorter overall route to the goal when travel around the current intermediate circle circumference is included. In order to determine which partial path which will be in

the shortest overall path, the two partial paths must be properly compared (Figure 7.8). An accurate comparison can be obtained by including the cost of the arc between the landing points of the two arriving partial paths. Application of Inequality (7.11) in Figure 7.8 indicates which partial path is shorter.

$$cost(path1) + \text{arc\_}cost(a, b, mode) \overset{?}{<} cost(path2) \qquad (7.11)$$

Note that such a comparison is only meaningful when each partial path follows the same direction of rotation around the intermediate circle being evaluated. The definitions of Circle and Arc explicitly include direction of rotation. Opposite rotation modes around the same obstacle are part of opposite circles and thus mutually exclusive paths. Should the leaving point for the actual shortest path lie between the two partial path landing points (e.g. points a and b of Figure 7.8), the longer of the arriving partial paths will correctly be excluded.

Successful determination of visibility and correct comparison of intermediate path costs for multiple points and circles allows the employment of search techniques to find the best path overall from start to goal. Formal statement and proof of the path comparison proposition can be found in Appendix F section 4.3 (Kanayama Brutzman 91).

D. ALGORITHM FOR DETERMINING VISIBLE TANGENTS

Assume that a general circle world is modeling n independent obstacles with $2n$ circles. Visibility determination for a single pair of points in this world has already been shown to have algorithmic complexity of order $O(n)$. The set of all pairs of points that may need to be checked includes line segments from the start point to the finish point, start point to a tangent on every circle, finish point to a tangent on every circle, and four mutual tangents between each possible pair of circles. Tangents are not possible between a circle and itself or its opposite. As a result, a total of $[4(2n)(2n - 2) + 4n + 1] = [16n^2 - 12n + 1]$ possible line segments may need to be evaluated.

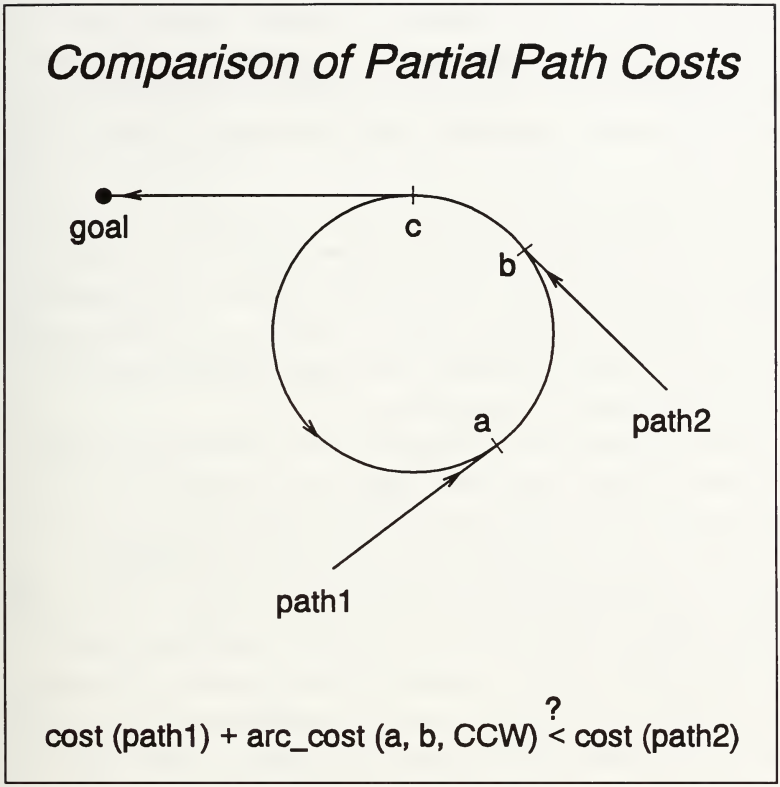


Figure 7.8 Comparison of partial path costs

Since there is no inherent geometric redundancy in any of these tangent points, each tangent segment must be considered independently of all others. However there is no requirement that every possible tangent segment be evaluated when determining the shortest path. The proper choice of which points and segments need to be evaluated is essential in order to conduct an efficient search.

Visible tangents from a single point to all other circles can be determined using a sweep method (Preparata 85) (Figure 7.9) in order O($n$log$n$) complexity. The point

to all circles visibility determination algorithm is elaborated in Figures 7.10 and 7.11. As tangents to each circle are evaluated, they are inserted into and removed from a heap in order to be sorted by orientation. A second pass of the sorted circle tangent data employs heap insertion and deletion for direct determination of visibility of each tangent.

Visible tangents from a single circle to all other circles (Figure 7.12 and Figure 7.13) can also be determined using this sweep method in order O($n$log$n$) complexity. The single circle to all circles visibility algorithm is nearly identical to the point sweep algorithm elaborated in Figures 7.10 and 7.11. Care must be taken to require proper matching of rotation mode values between sweep direction and the sweep circle when implementing the algorithm.

Similar sweep method algorithms have been used for determining line segment visibility (Welzl 85) and polygon visibility (Asano 85) in order O(n^2) time. It is interesting to note that substitution for each circle by a line segment connecting the left and right tangents to that circle appears to make this type of sweep nearly identical to the individual line segment visibility sweep in (Welzl 85). However, each individual circle has order O(n) such tangent landing and leaving points associated with it. Thus, a circle world has at least order O(n) more segments in the complete tangent visibility graph than a polygonal world or line segment world. This increased complexity is a significant difference between the geometry of the circle world problem and the standard polygon-based configuration space treatments referenced previously.

E. SHORTEST-PATH DIJKSTRA AND A* SEARCH ALGORITHMS

Dijkstra's algorithm is a standard approach to solving the single-source shortest-path planning problem (Manber 89). Given a start point in a circle world, proper application of Dijkstra's algorithm will calculate the shortest path to each clockwise and counter-clockwise circle until the designated goal point is reached. This is a greedy algorithm in that the shortest available path from the start point is always

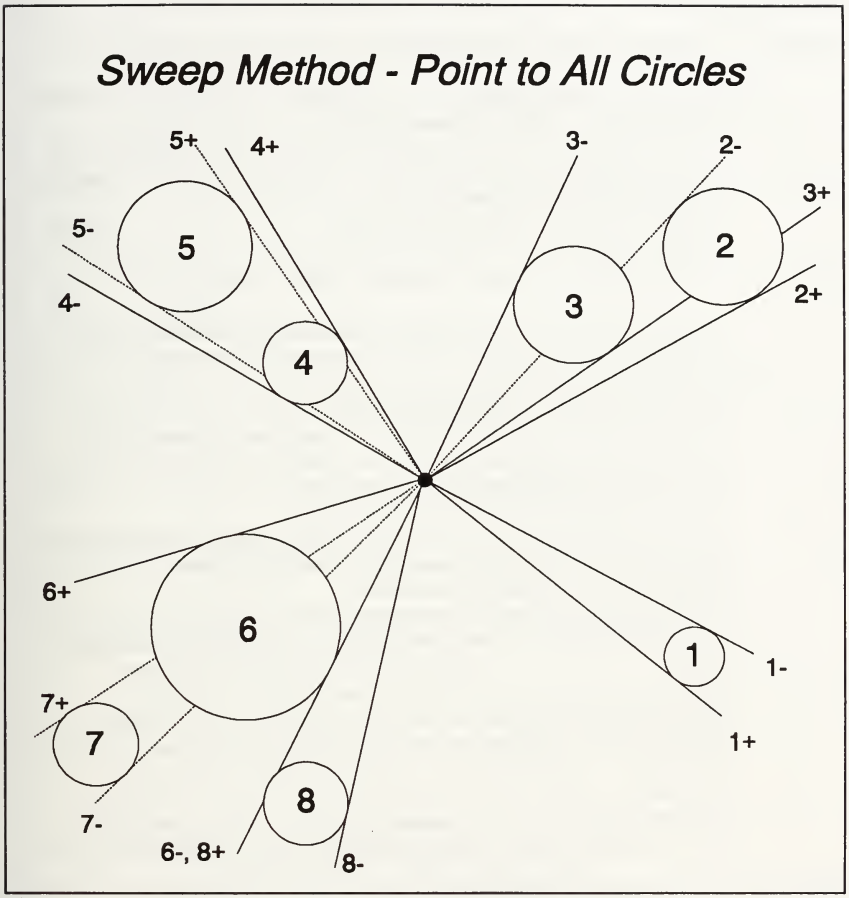


Figure 7.9 Sweep visibility determination from point to all circles

selected upon each iteration. Essentially the algorithm branches outward equally in all directions.

While this algorithm is useful for building a complete collection of shortest paths to all circle obstacles, it is extremely inefficient if only the shortest path to a single

O(*n*log*n*) Calculate and sort circle tangents:

-- Calculate left and right tangents to each circle. Note that CCW and CW rotations denote separate independent circles.
-- After calculating a circle's tangents, use heap insertion to sort the circle relative to others by orientation. If two orientations are equal, secondary sort key is shortest tangent distance.
-- Keep track which is shortest right tangent.

O(*n*log*n*) Perform circular sweep to determine tangent visibilities:

-- Initialize sweep termination angle equal to sweep starting angle.
-- Starting at circle with shortest right tangent, conduct a complete sweep in counter-clockwise direction.
-- Meet right tangent: insert circle into heap.
-- If circle inserted is on top of heap (or tangent distance equals top of heap), mark that right tangent as visible. Otherwise if circle was below top of heap, mark right tangent as nonvisible.
-- Meet left tangent: remove circle from heap.
If removed circle was on top of heap (or tangent distance equaled top of heap), mark that left tangent as visible. Otherwise if circle was below top of heap, mark left tangent as nonvisible.
-- If circle with current left tangent was not found in heap, it overlaps the sweep starting angle. Change sweep termination angle to current sweep angle.
-- Update circle heap pointer cross-reference table.
-- Similarly process any other circle tangent(s) at current sweep angle.
-- Increment sweep angle to next circle in sorted circle table and repeat for all circles.
-- If necessary, resume at start of circle table and continue sweep until all tangents through the sweep termination angle are reprocessed. This allows any circles which overlap the start angle to be correctly evaluated.

Figure 7.10 Explanation of sweep visibility algorithm from point to all circles

goal point is desired. If the robot is repeatedly operating in a fixed environment and must frequently return to a specific location, however, a single application of Dijkstra's algorithm using that specific location as a start point will provide all shortest paths past all fixed circle obstacles. Precalculation of all shortest paths of

```
Algorithm Sweep_Visibility_from_Point (point, circle_world);

Input:  Point from which visibility is to be checked, and circle_world which
        includes start point, goal point, and both CW and CCW rotations of all
        circles.
Output:     All cross-tangents and external tangents originating at point, including
            lengths, endpoint coordinates, orientations and visibilities.
begin
n := 2 (#circles) + 2;  {include CW/CCW circles, start & goal points}

for i := 1 to n do
  tangent := calculate tangent (point, circle [i]);
  push (tangent, tangent_heap);
        {primary sort key is orientation, secondary sort key is tangent length}
  if tangent.length < shortest_tangent.length then
     shortest_tangent := tangent;

for i := 1 to n do   sorted_tangents [i] := pop (tangent_heap);

end_angle:= shortest_tangent.orientation;
i        := shortest_tangent index in sorted_tangents list;

while    sorted_tangents [i].orientation precedes or equals end_angle do
  if sorted_tangents [i].mode = RIGHT then
        push (sorted_tangents [i], circle_heap);
        {sort key is length}
        if (top (circle_heap).distance = sorted_tangents [i].distance) then
              sorted_tangents [i].visible := VISIBLE;
        else  sorted_tangents [i].visible := NONVISIBLE;

  if sorted_tangents [i].mode = LEFT then
        if not found (sorted_tangents [i], circle_heap) then
             end_angle := sorted_tangents [i].orientation;
        {continue sweep until all circles fully evaluated}
        else if (top (circle_heap).distance = sorted_tangents [i].distance) then
             sorted_tangents [i].visible := VISIBLE;
        else  sorted_tangents [i].visible := NONVISIBLE;
        pop  (sorted_tangents [i], circle_heap);
        i := (i + 1) mod (n + 1);  {look at next circle in ordered list}
end
```

Figure 7.11 Pseudocode for sweep visibility algorithm from point to all circles

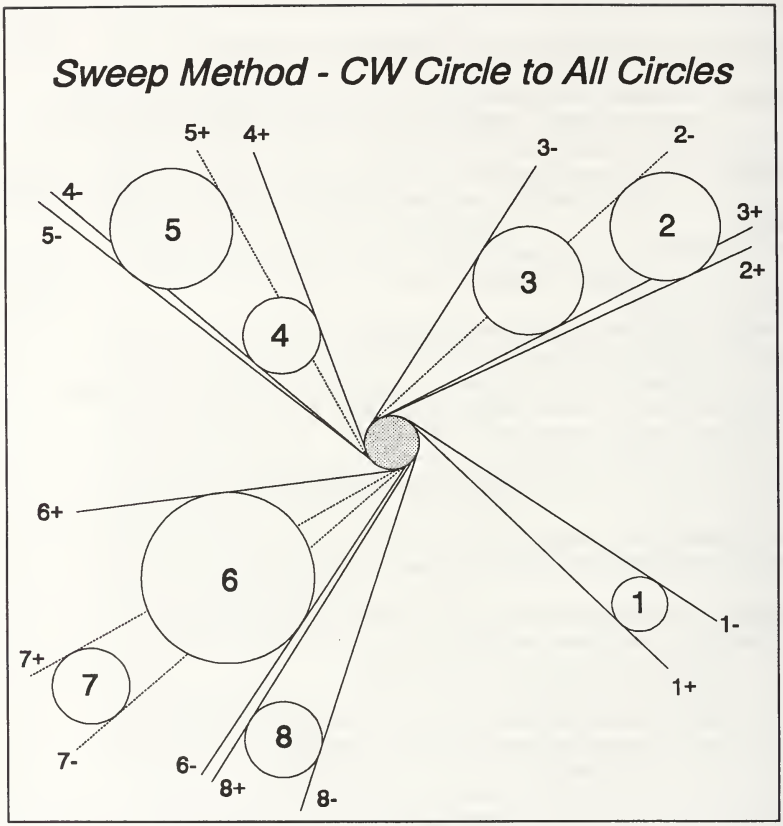


Figure 7.12 Sweep visibility determination from clockwise circle to all circles

interest may be more cost-effective than recalculating the latest shortest path objective immediately prior to travel.

In the case of circle world search, the A* search algorithm overcomes the inefficiencies of Dijkstra's algorithm by applying an evaluation function to each iterative selection in order to proceed as directly as possible towards the goal. Distance remaining to the goal and arc cost around the current circle obstacle are

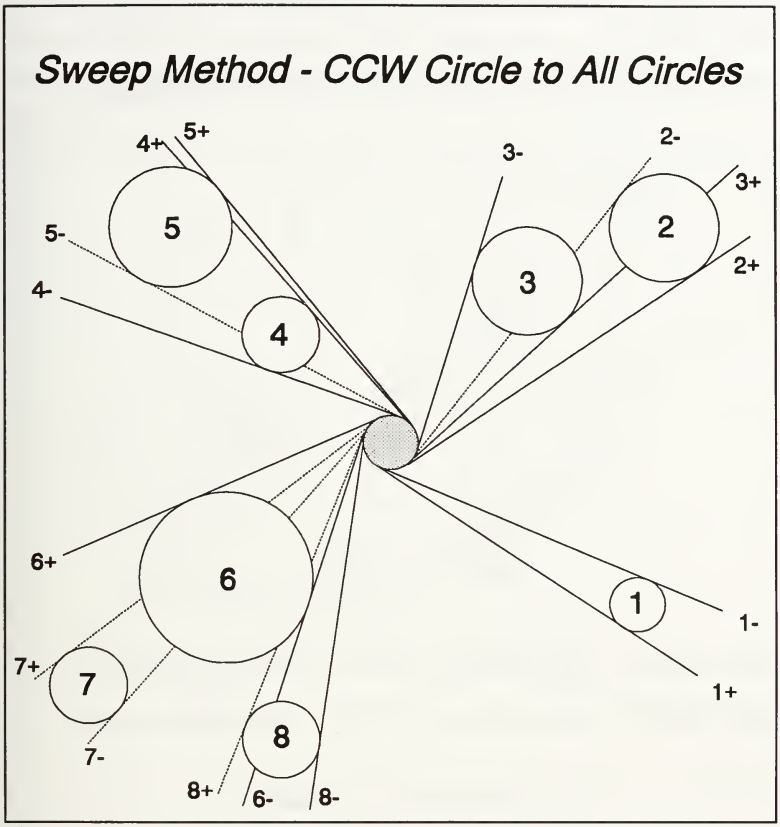


Figure 7.13 Sweep visibility determination from counter-clockwise circle to all circles

included with path cost when determining which path of the many available should be extended next (Figure 7.14). This search approach has the effect of driving the search in the direction of the goal and ignoring paths that are expensive, i.e. paths that are heading in a wrong direction.

The evaluation function comparison test applied in a circle world search is given in Inequality (7.12) and Figure 7.14. Note that the evaluation function term for

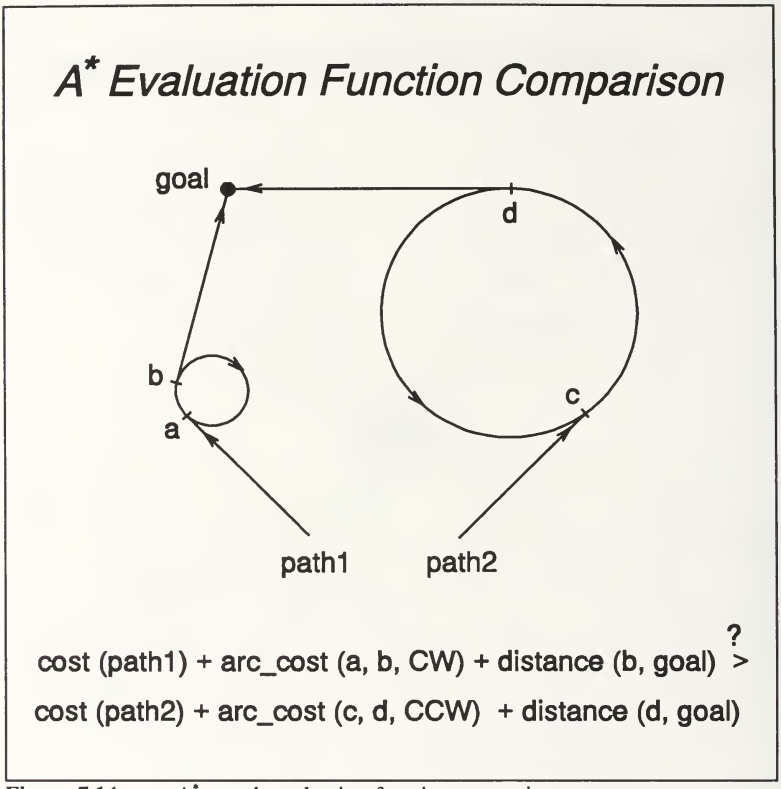


Figure 7.14 A* search evaluation function comparison

projected tangent segment distance from the current circle to the goal point does not require the projected tangent segment to be visible.

$$cost(path1) + \text{arc\_}cost(a, b, mode) + distance(b, goal) \overset{?}{\geq} cost(path2) + \text{arc\_}cost(c, d, mode) + distance(d, goal) \qquad (7.12)$$

As expected, the A* search algorithm is much more efficient than Dijkstra search in a circle world. Given a reasonably uniform distribution of *n* circle obstacles about a start point, Dijkstra's algorithm tends to reach the goal in order O(*n*) search steps (Figure 7.15).

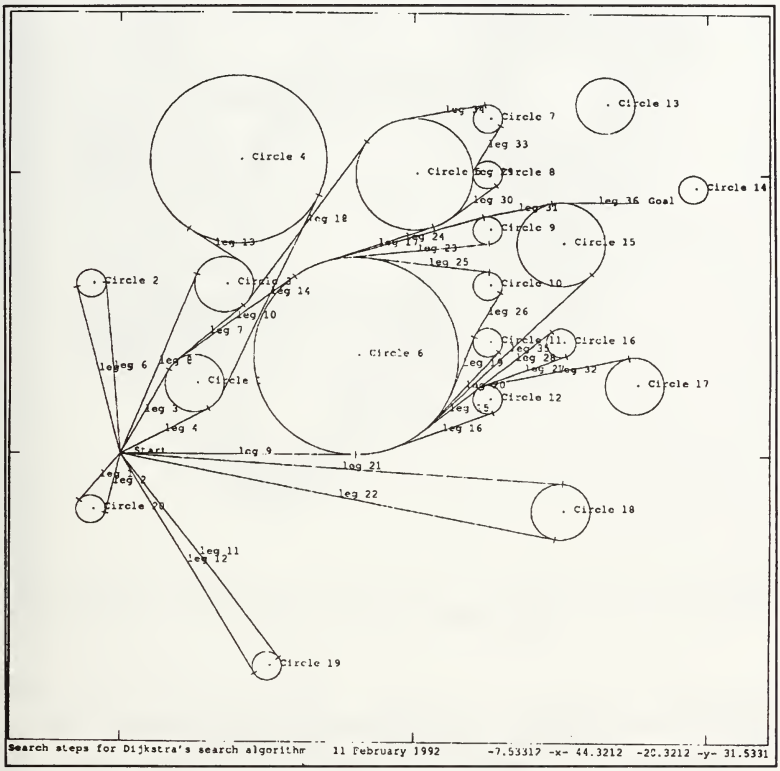


Figure 7.15 Search steps displayed for Dijkstra's search algorithm

It is conjectured that an A* search requires only order O(log*n*) search steps for the same uniformly distributed circle world (Figure 7.16). Algorithmic efficiency is a critical consideration in the path planning problem, since the total number of tangent

segments in the visibility graph is quite large. For the twenty circles of the challenging example circle world, 684 of 1682 total tangents (40.7%) are visible (Figure 7.17).

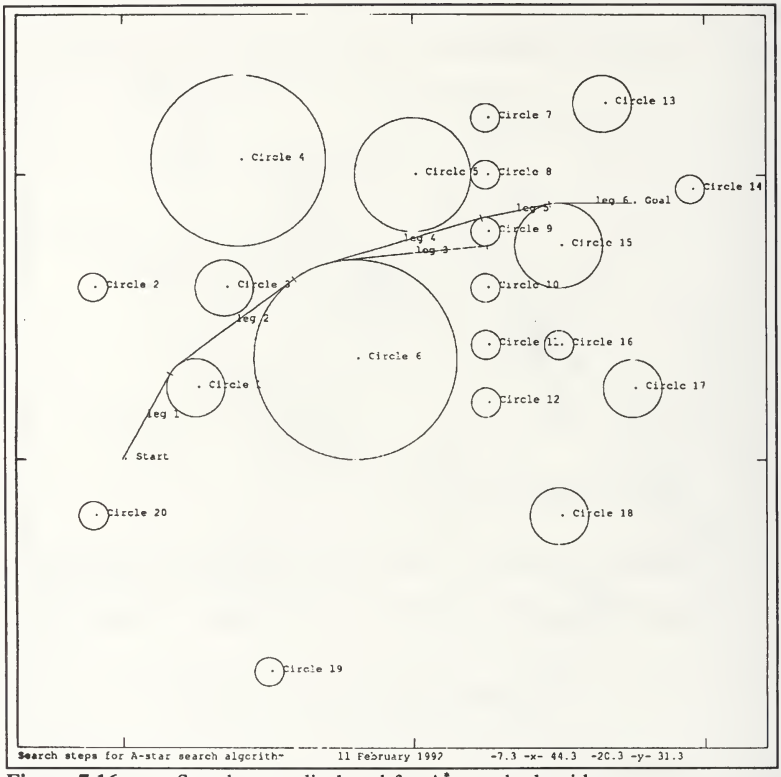


Figure 7.16 Search steps displayed for A* search algorithm

Worst case A* search approximates Dijkstra algorithm performance, approaching order $O(n)$ steps only when all obstacles are between the start point and the goal. Thus in every case A* search algorithm performance is superior to the Dijkstra algorithm. A formal proof regarding the optimality of the A* search method in graph

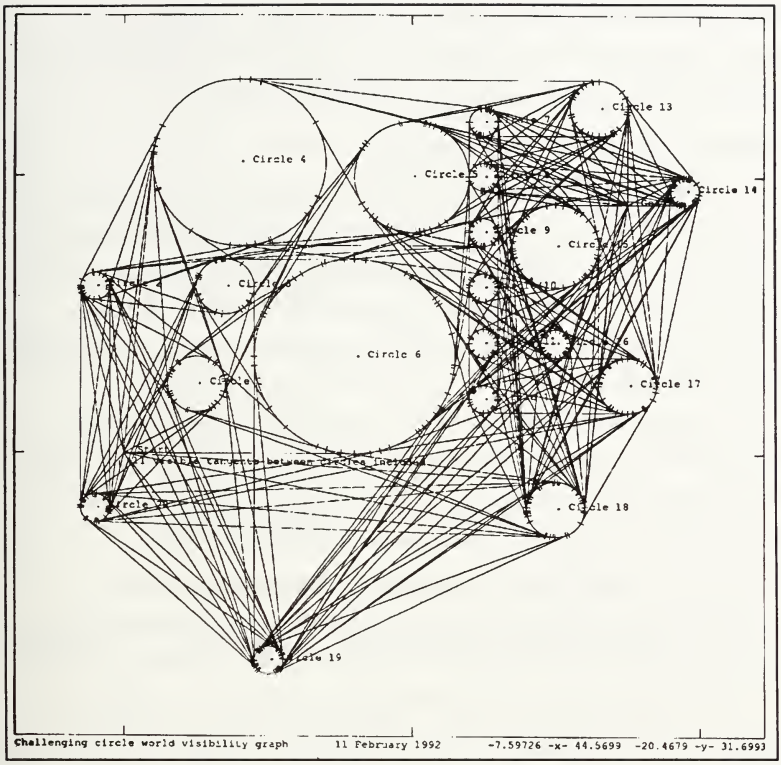


Figure 7.17 Challenging circle world visibility graph

searching can be found in (Hart 68).

The circle sweep algorithm (Figures 7.10 and 7.11) calculates visibility from a point or circle to all other circles in order O($n$log$n$) complexity. The algorithmic complexity of the complete path planning problem using A* search is therefore order O(n^2log$n$) in worst case, and conjectured to be order O($n$log^2n) in best case. It is important to note that the best case is the most likely to occur situation where all circles are distributed randomly. The worst case is the less likely situation where all

circle obstacles lie directly between the start and goal points. Additional work is needed to formally determine the complexity of A* search relative to Dijkstra search given the special geometric constraints of a general circle world.

F. IMPLEMENTATION AND RESULTS

The author has written a program in ANSI C to illustrate and evaluate the methods described in this paper. The full program consists of geometric primitive data structure definitions, a library of spatial reasoning functions, tangent visibility determination routines, search functions to perform either Dijkstra or A* search, graphical screen and hard copy output, and a text-based user interface. Source code is included as Appendix F.

Two methods are employed for output. Primary output consists of large coordinate text files including embedded labels (Figures 7.15, 7.16 and 7.17). These files can be sent to create two-dimensional graphical output (Figure 7.18) using plotting routines such as the Unix *graph* command or the *sunplot* routine (Mullender 87). Printed hard copy or screen graph output can be generated.

Secondary output consists of a listing of geometric primitives such as circles, points, paths, segments and arcs generated during execution. These high-level text outputs can be easily used as input for revised circle world search runs (Figure 7.19). The outputs can also be passed as input to independent robot motion control processes or graphical simulation programs such as the NPS AUV Integrated Simulator.

ANSI C was chosen as the project programming language for purposes of efficiency, portability and compatibility with other robot applications at NPS. It is important to note that circle world program inputs and outputs are language and platform independent, however. While the circle world programs normally run on powerful workstations under the Unix operating system, they are completely portable to other architectures such as the IBM PC or GESPAC 68030/OS-9 running on the NPS AUV.

```
 0.000000 0.000000
". Start"
 35.000000 18.000000
". Goal"
 5.000000 5.000000
". Circle 1"
 7.000000 5.000000
 6.999695 5.034905
 6.998782 5.069799
[.... etc. around each circle perimeter]
 7.000000 5.000000
 " "
 16.500000 5.000000
"Straight line start to goal (cost = 34.48) "
 0.000000 0.000000
 16.000000 0.000000
" "
 8.000000 0.000000
"      Best path start to goal (cost = 36.11) "
 16.000000 0.000000
" "
[.... various line segments & bounding points follow]
```

Figure 7.18 Excerpt from graphics plot file intermediate output

The current circle world implementation program uses an exhaustive and iterative visibility determination algorithm instead of the more efficient sweep method algorithm presented here. Source code optimization is worthwhile but not mandatory prior to integration on board an operational real-time vehicle such as the NPS AUV.

G. THREE-DIMENSIONAL APPLICATIONS AND FUTURE WORK

The two-dimensional approach to path planning provided by the circle world model can be directly used for robot motion on any planar surface, such as a laboratory, shop or warehouse floor. Interestingly, the circle world approach is not constrained to path planning in two dimensions. This extendability is a valuable result since efficient path-planning algorithms in three dimensions are considered to be an

Circle_World Shortest Path Determination

Point	10.00	5.00	0.00	Start
Point	125.00	65.00	0.00	Goal
Circle	10.00	25.00	0.00	5.00
Circle	40.00	30.00	0.00	20.00
Circle	75.00	20.00	0.00	15.00
Circle	100.00	45.00	0.00	18.00
Circle	20.00	60.00	0.00	5.00
Circle	115.00	10.00	0.00	5.00
Circle	65.00	55.00	0.00	5.00

Path planning results: Best path (cost 143.8)

Segment	5.00	5.00	0.00	42.72	10.19	0.00	
Arc	40.00	30.00	0.00	20.00	277.83	328.11	1=CCW
Segment	56.98	19.43	0.00	62.26	27.92	0.00	
Arc	75.00	20.00	0.00	15.00	148.11	139.87	-1=CW
Segment	63.53	29.67	0.00	86.24	56.60	0.00	
Arc	100.00	45.00	0.00	18.00	139.87	94.45	-1=CW
Segment	98.60	62.95	0.00	125.00	65.00	0.00	

Figure 7.19 High-level text listing of example NPS pool circle world and shortest path determination

area where more research is needed (Yap 87).

A circle world can be used for robot path planning across irregular land terrain. The fact that such terrain may not be level is not limiting as long as the robot can generally traverse it. Only vertical obstructions, interfering holes and excessive slope variations that prevent safe robot passage need be modeled as circle obstacles. Remaining terrain surface features can be treated as planar and part of the obstacle-free portion of the configuration space.

Three-dimensional path planning by robot vehicles can typically be performed using a similar circle world approach. Most three-dimensional obstacles can be represented as circles by taking the cross-section of each obstacle on a level plane

used for robot travel (Figure 7.20). An irregular object can then be modeled using a vertical cylinder. Such a cylinder defines the minimum radius circle needed to enclose all portions of the object which are collision threats in the plane or region of robot travel. Underwater obstacles can be modeled in a similar fashion, where the circle world ground plane represents the allowable depth band of robot submarine travel.

These circle world conditions are necessary and sufficient to model any set of real world obstacles as long as robot motion is along a plane or within a planar region. Three-dimensional obstacles are represented as cylinders, and a path planning search through such a three-dimensional obstacle space is reduced to a two-dimensional circle world search. Such a representation is particularly natural for underwater obstacles. An example set of underwater obstacles in the NPS pool is shown with a corresponding shortest path in Figure 7.21, which is a direct output of the circle world path planning program. The same data set is also shown in Figure 7.22 as rendered by the NPS AUV Integrated Simulator using high-level text output from the circle world path planning program.

These circle world methods are extendable to an analogous approach for polygon world modeling. Additional program coding is needed to address overlapping and adjacent obstacles. The simplicity, efficiency and effectiveness of the circle world model makes it well suited for real-time path planning by mobile robots.

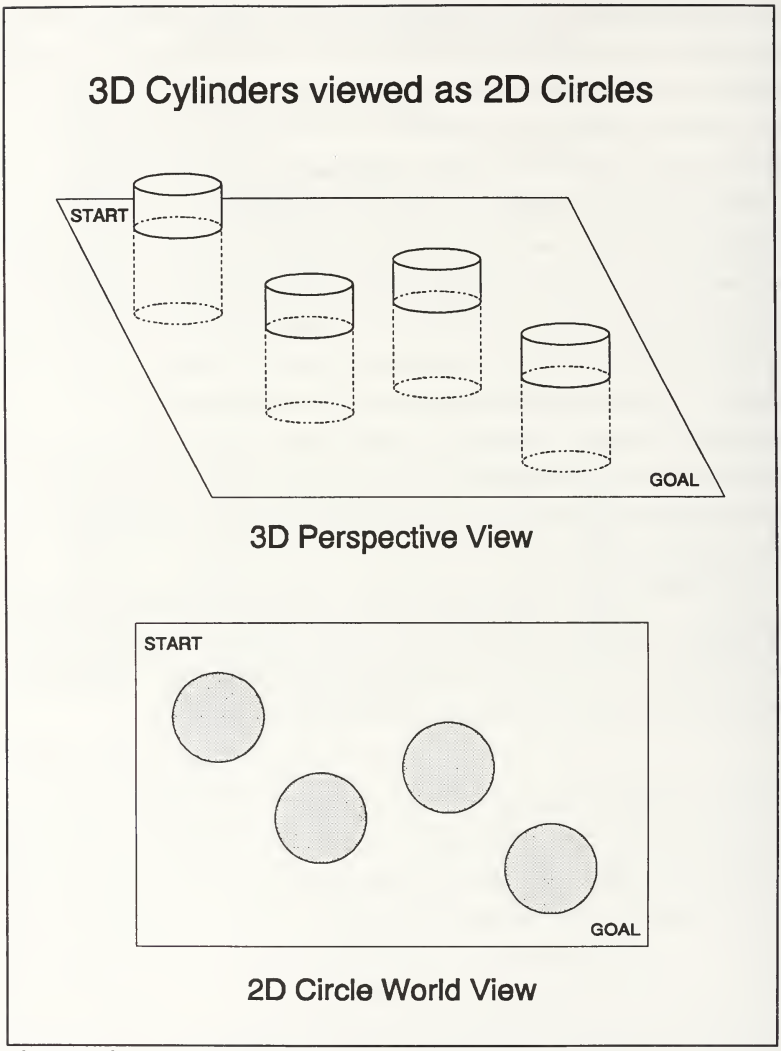


Figure 7.20 Three-dimensional cylindrical obstacles viewed as two-dimensional circles

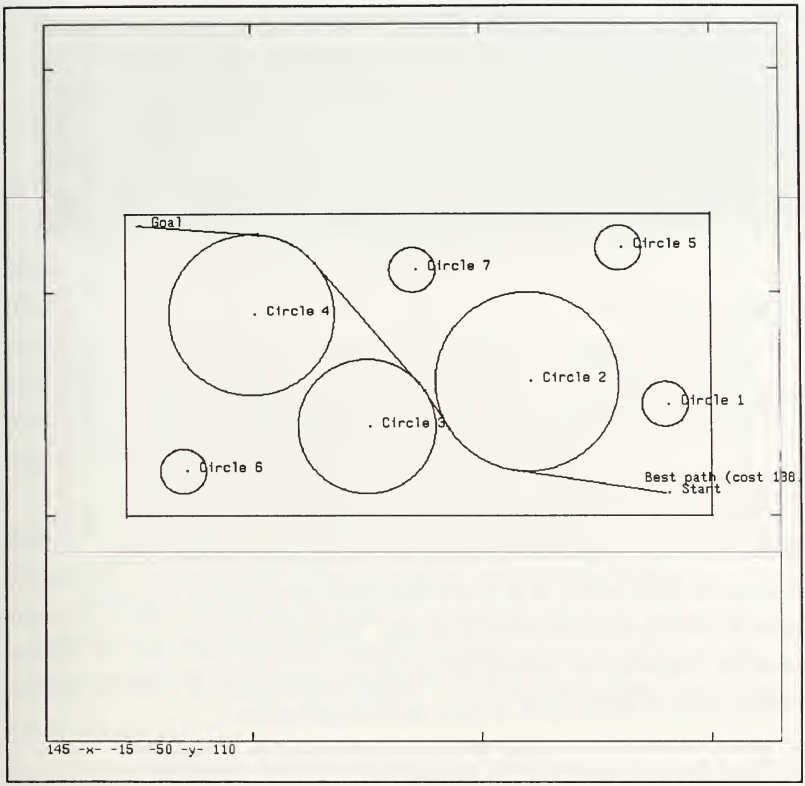


Figure 7.21 Two-dimensional representation of obstacles in the NPS pool

Figure 7.22 Three-dimensional representation of obstacles in the NPS pool

H. CONCLUSIONS

Mobile robot path planning around obstacles can be accomplished by modeling obstacles as pairs of circles with opposite rotations. Addition of robot radius and safety standoff distances to circle radii allows modeling the robot as a point. This circle world model can be used to calculate shortest paths between points.

Tangents between circles in a circle world have no inherent redundancy or duplication due to the uniqueness of landing and leaving points on each circle. Alternate partial paths landing at an intermediate circle obstacle must be properly compared in order to determine which is shortest. Dijkstra's algorithm or (preferably) A^* search can selectively use visibility and partial path comparison calculations to find the shortest, safest or optimal path between start and goal points.

Tangent visibility from a single point to all circles can be calculated in order $O(n\log n)$ time. Similarly, tangent visibility from a single circle to all other circles can be calculated in order $O(n\log n)$ time. The shortest path between start and goal points can be calculated in order $O(n^2\log n)$ time.

Obstacle avoidance is a typical robot behavior regardless of obstacle height. The circle world search model is directly extendable to the general case of three-dimensional path planning and is particularly suitable for underwater vehicle path planning. Future circle world path planning implementations on a robot vehicle should include polygon obstacles as well as overlapping and adjacent obstacles.

VIII. REAL-TIME OPERATING SYSTEM AND AUV SIMULATION CONSIDERATIONS

A. NPS AUV AND REAL-TIME OPERATIONS

The NPS AUV is an untethered robot submarine designed for research in adaptive control, mission planning, mission execution, and post-mission data analysis (Healey 90). AUVs are typical of other autonomous robots in that a large number of internal processes must run simultaneously while meeting stringent real-time requirements. AUVs differ from other robots in that they are designed to operate submerged and isolated from communication or external directions. Such missions require extraordinarily reliable and robust vehicle performance.

The principles and concepts particular to real-time operating systems are clearly defined in a wide variety of references (Blackman 76) (Mellichamp 83) (Deitel 90) (Nelson 92). An explanation of real-time control issues directly relating to autonomous robot vehicles can be found in an excellent case study comparison between the Ohio State University Adaptive Suspension Vehicle (ASV) and the Defense Advanced Research Projects Agency (DARPA) Autonomous Land Vehicle (ALV) (Payton 91).

Although numerous software modules have been written with the NPS AUV in mind, very little software has actually been implemented, integrated or tested underwater in real time. The main reason for this deficiency is the current lack of a flexible high-level software control module that can efficiently coordinate multiple NPS AUV processes using the OS-9 real-time operating system.

This chapter examines the real-time operating system issues that are pertinent to the development of the NPS AUV. These considerations pertain equally to the Gespac/OS-9 microprocessor and support hardware portions of the NPS AUV Integrated Simulator.

B. HARD AND SOFT REAL-TIME REQUIREMENTS

In order to perform numerous sophisticated mission functions, multiple processes must be operating simultaneously while meeting both strict and relaxed real-time schedule requirements. The autonomous nature of an AUV requires operation without external backup in a harsh and unforgiving environment. Vehicle control, sensor evaluation, underwater navigation, search, path planning, obstacle avoidance, fault tolerance, and numerous other processes are required. All processes must interact with the external environment and each other in real time with varying degrees of interdependence (Bobrow 91).

It is important to distinguish between hard and soft scheduling criteria for real-time processes. Mission-critical actions such as vehicle control and failure detection are hard real-time scheduling requirements. Failure to meet such hard deadlines may result in mission failure or even catastrophic loss of the vehicle. Conversely, high level logical processes such as path planning or mission replanning might always be considered soft requirements, since their execution is rarely mandatory for safe vehicle operation and immediate results are not required. Finally, some processes may have priorities that vary from soft to hard depending on circumstances. For example, obstacle avoidance is typically a soft requirement until target proximity or rapidly closing range rate make immediate action necessary to avoid collision.

C. NPS AUV PROCESS DEADLINE SPECIFICATION AND SCHEDULING

The current NPS AUV mission software schedule runs a single simple mission control loop using a 10 Hz clock. A full 100 millisecond interval is allotted for each mission loop, but no processes are allowed to exceed that period. This interval is adequate to perform numerous tasks: compute basic vehicle control orders, transmit using one to four sonar transducers, record all current vehicle data parameters in working memory, perform rudimentary sonar analysis, detect waypoints, detect potential collision, and order predetermined state changes in propeller speed and

control surface position. Typically very little time remains at the end of each fixed 100 millisecond time segment. Such a simple hard-wired timing mechanism is not a feasible control architecture for AUV software of even slightly greater complexity.

Proposed NPS AUV software modules are shown by the data flow diagram of Figure 8.1 (Healey 90). Only the basic interdependencies of these NPS AUV task modules have been characterized. A formal analysis of software module specifications, timing requirements, task periodicities and concurrency dependencies has yet to be performed. It is likely that the NPS AUV software modules will ultimately have timing constraints and periodicity characteristics similar to those developed in Table VIII.1.

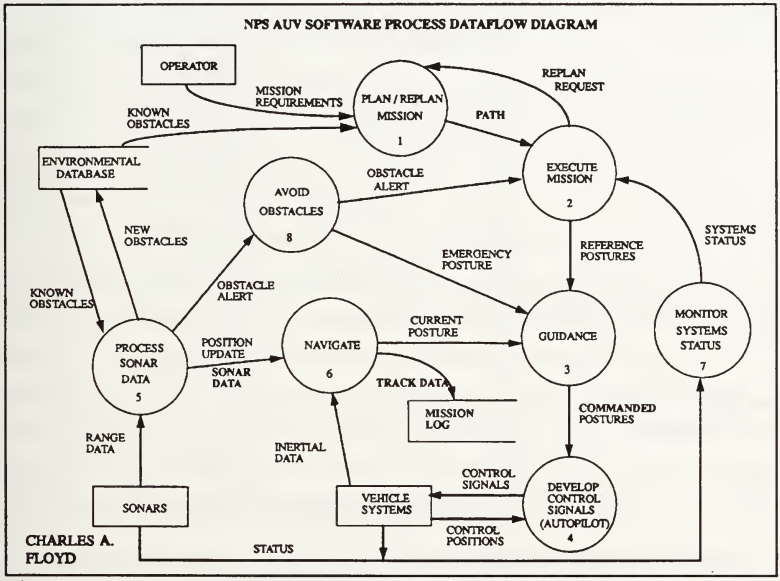


Figure 8.1 NPS AUV software process dataflow diagram

Current real-time operating system research at NPS has focused on rate-monotonic scheduling theory, an approach that employs fixed prioritization of

Table VIII.1 AUV SOFTWARE MODULE REAL-TIME CHARACTERISTICS

AUV Software Modules	Real-Time Execution Characteristics		
Plan/Replan Mission	Soft	Aperiodic	Event triggered
Execute Mission	Hard	Periodic	Every control loop
Guidance	Both	Periodic	Every loop unless preempted
Autopilot	Hard	Periodic	Every control loop
Process Sonar Data	Hard	Periodic	Every loop unless preempted
Navigate	Soft	Aperiodic	Every loop unless preempted
Monitor Systems Status	Both	Periodic	Every loop unless preempted
Avoid Obstacles	Hard	Periodic	Every control loop

processes and guarantees acceptable average performance (Leatherman 91) (Makris 91). Benefits of rate-monotonic scheduling include guaranteed completion of periodic tasks in order of priority, fast response for aperiodic tasks, modifiable task priorities, and the scheduling of tasks that permit imprecise computations (i.e. output

precision proportional to time available). Rate-monotonic task analysis and scheduling is performed off-line prior to actual execution of system software. Rate-monotonic scheduling implementations running dummy processes under the OS-9 operating system have been shown to provide processor utilization above 80% and graceful degradation under overload.

Dynamic adjustment of constraints and process schedules may ultimately be required to ensure successful AUV operation during unforeseen tactical scenarios or pathological process conflicts. Dynamic scheduling theory requires further formal research to provide a verifiable theoretical foundation, but it appears to be a desirable model for the distributed artificial intelligence applications likely to make up the AUV. In this regard the FLEX programming language is worth consideration since it implements dynamic scheduling theory and generates C++ code as output (Kenny 91). Additionally there exists a hybrid approach known as a mixed priority system that combines the best features of rate-monotonic scheduling and dynamic scheduling (Leatherman 91). Formal evaluation of the mixed priority approach also appears worthwhile.

Given that AUV-related research is likely to continue for many years by several academic departments at NPS, operational software changes and additional new software processes will always be under development and require integration into the overall NPS AUV system process schedule. Reliability, compatibility and extendibility for future growth must be key requirements for any proposed control process timing schedule. Robust and flexible interactions between numerous interdependent processes will be essential to allow frequent improvements to vehicle performance while maintaining vehicle reliability.

D. PARALLEL PROCESSING AND CONCURRENT PROGRAMMING

It is important to note that parallelism is equally as important as real-time scheduling for an AUV operating system. This is particularly true if low-level control, complex behaviors, sensor fusion, data analysis, mission planning and numerous other

artificial intelligence aspects of robot mission execution must all coexist and cooperate in a rapid manner (Stankovic 88).

Non-trivial robot performance requires that numerous processes operate in parallel, either independently or in a mutually dependent fashion (Kasahara 88). Numerous challenging AUV mission requirements will inevitably lead to multiple software modules operating concurrently. Such parallelism might be most easily implemented using a multiprocessor architecture. An AUV's real-time operating system must completely support concurrency constructs that are fully integrated with the real-time scheduling mechanisms.

Several standard features of parallel programming are necessary for effective software engineering of a real-time AUV. Adequate shared memory is essential if numerous processes are to quickly and efficiently access system state variables and the large amounts of time-sensitive sensor data that is expected. Predictable rendezvous, synchronization and communication methods must be available, both for interaction between mutually dependent processes as well as loose overall control by a mission executor module.

Increasing hardware sophistication can further allow tasks to be distributed over a network among separate specialized embedded processors (Stankovic 88). Extensions of process rendezvous, synchronization and communication must be available for distributed processing if networked processors are to be employed.

Massively parallel processing in the classic sense uses numerous processors in parallel to perform array processing or numerous parallel solutions of identical algorithms. Such an approach is not a likely requirement for AUV operation. Aside from potential analysis of sophisticated sensor data using vision processing techniques, few (if any) AUV functions can be decomposed into numerous identical subproblems. The diverse nature of the many AUV software modules implies that a transputer architecture is not a prerequisite for successful integration of multiple AUV processes. Nevertheless the transputer paradigm may be an effective way to minimize interface difficulties while allowing unlimited addition of numerous unique parallel processes

under a single integrated real-time operating system. This approach is also being considered by C.S. Draper Laboratories for the next-generation software architecture of the DARPA Unmanned Underwater Vehicle (UUV) (Hale 91).

E. OPERATING SYSTEM COMPATIBILITY AND INTEROPERABILITY

It is important that the AUV operating system be fully compatible with all current and projected vehicle hardware and software. External connectivity of the real-time operating system is also important.

Hardware interoperability considerations must consider connections between multiple processors of various types internal to the vehicle, as well as numerous analog/digital and digital/analog interfaces. Space, weight and power requirements are very strict so internal AUV hardware architectures must be closely compatible. Physical compatibility improves vehicle endurance by reducing power consumption.

Software compatibility is less critical than hardware compatibility, but software incompatibilities can still impose undesirable processing delays if too much work is required to translate communications between processors. Network support and software interfacing will be needed when different operating systems reside on multiple processors. Multiple programming language support is desirable for unrestricted research in a variety of control system and artificial intelligence subjects. Process encapsulation is desirable in order to minimize faults and side effects during software development. High-level software access to machine-dependent, machine-level and device-dependent routines is also needed. Such routines permit various processes to utilize the operating system for direct access and control of the numerous physical components of the AUV.

Although an AUV is untethered and isolated during operation, a number of external compatibility requirements remain. Mission data collection, consolidation, storage and transmission are ultimately targeted for external off-line post-processing and analysis. Distributed processing over a network internal to the AUV requires that each individual operating system must be able to interact with the others. Interactive

network communication is also a likely requirement for on-line laboratory testing of the AUV. Connecting the vehicle or similar lab prototypes to an integrated simulator allows scientific visualization of AUV processes for active real-time end-to-end developmental testing. For these reasons the AUV must be able to communicate in some fashion with non-native operating systems and software environments. External connectivity is essential to support the diverse and distributed communities that conduct AUV research.

Several other operating systems are worth noting. Real-time constructs and compatibility can be incorporated into typically non-real-time operating systems such as Unix by adding specially designed message-passing processes (Cramer 88) (Falk 88) (Hildebrand 88). Modified real-time kernels of common operating systems such as Modular Computer Systems Inc.'s *Real/IX* for Unix or Digital Research Inc.'s *FlexOs* for DOS are viable and commercially available (Falk 88) (Baerson 91). Standards development work continues for Posix, an open operating system specification based on Unix that includes real-time constructs (Deitel 90) (Falk 88). As robotic systems and intelligent machines become more commonplace, the interactive design concepts of TRON (The Real-time Operating system Nucleus) will become increasingly important (Kahaner 91). Finally, a distributed operating system may provide the most efficient control mechanisms for distributed processors sharing distributed resources (Dasgupta 91).

F. OS-9 OPERATING SYSTEM

OS-9 is Microware System Corporation's real-time operating system used by the NPS AUV. OS-9 is designed to run exclusively on the Motorola 68020/68030/68040 family of microprocessors (GESPAC 89). The two specific hardware configurations used in the NPS AUV include GESPAC 68020 or 68030 microprocessors connected to a GESBUS (VME bus compatible) backplane. Serial ports, parallel ports, analog/digital interface cards and an Ethernet interface are available for internal and external connections. Also available for internal networking is an Intel 80386

microprocessor. INMOS T805/T425 transputers were also to be connected but are no longer manufactured in a configuration compatible with the GESBUS. No suitable transputer replacement has yet been identified.

The OS-9 process states available include start (fork), active, run, exit, sleep, wait (process synchronization), and event wait (semaphore communication). Process state transitions are shown in Figure 8.2 (GESPAC 89).

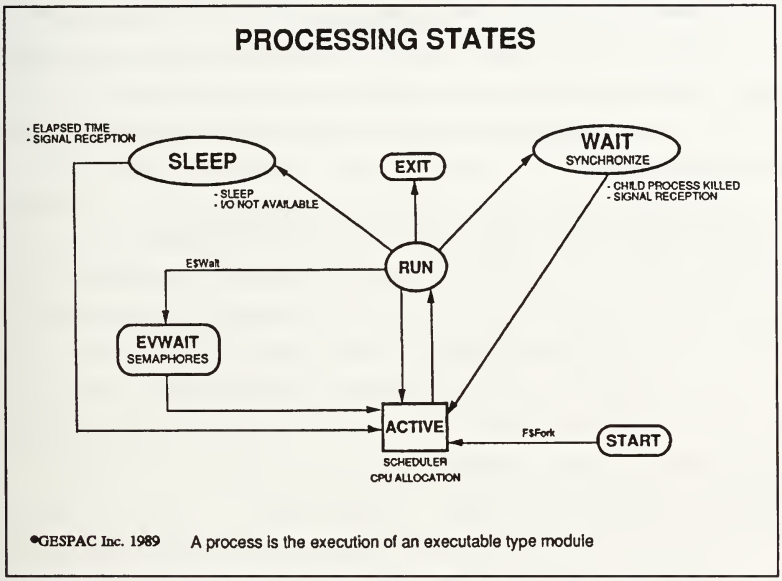


Figure 8.2 OS-9 operating system process states

OS-9 features that support expected AUV operating system requirements include adjustable priorities and aging for explicit execution scheduling, preemptive process switching based on priority, reprogrammable interrupts, a trap library, events for process synchronization, signal communication between processes, pipes for interprocess data transfer, and redirection of process inputs and outputs (Dibble 88).

Identical syntax when referring to processes or device drivers is a particularly convenient feature of OS-9.

Deficiencies and shortcomings of OS-9 include the current lack of compatible Ada or C++ compilers and no simple method of deadlock protection. Additionally a fully modifiable operating system kernel must be prepared through careful EEPROM configuration prior to operation. This preparation allows setting up the operating system to include only the device drivers that are necessary for the current vehicle hardware. OS-9 is very flexible in that additional drivers may be loaded at any time after system initialization by software command. However the EEPROM configuration process is time consuming, version dependent and error prone due to limited self-diagnostic testing.

G. CURRENT PROBLEM AREAS AND FUTURE RESEARCH

Deadlock detection in a real-time vehicle can be guaranteed by designing a special periodic real-time process for that purpose. An example can be shown using the NPS AUV software module information in Figure 8.1 and Table VIII.I. The NPS AUV has a tight inner control loop that includes the Mission Executor and Autopilot that must completely repeat on a frequent periodic basis of approximately one second. These two periodic processes can be required to toggle state variables every time the one-second control cycle is successfully completed. Failure to do so after several seconds is a clear indication of some type of critical problem such as deadlock. Recovery after deadlock detection can be promptly accomplished by reinitializing vehicle control loop software. This new approach to real-time deadlock detection is a straightforward solution to a problem that is frequently considered intractable in non-real-time operating systems.

Deadlock prevention is expensive but essential because the independent, unmonitored and uncontrolled nature of an AUV makes reliability paramount for vehicle survivability. Redundant approaches to deadlock prevention, deadlock detection and deadlock recovery are worthwhile. Resolution of deadlock is a

particularly sensitive area, given the frequently changing NPS AUV software and the unpredictable ordering of process preemptions and interactions in real time.

As various software modules are integrated into the NPS AUV, software engineering considerations become increasingly important. Key issues are systems integration, verification and validation of process behaviors despite real-time interaction uncertainties, software version control, and system software upward compatibility for integration of future software modules. Failure to methodically address software engineering issues will undoubtedly lead to unpredictable AUV behavior and tremendous amounts of time wasted troubleshooting individual software modules rather than subtle faults in the operating system implementation.

Fault tolerance is also needed to guarantee overall vehicle reliability and robustness. The approach taken needs to primarily rely on software checks, rather than the use of redundant processors found in some larger vehicles (Hale 91). Fault tolerance requirements will need to be specified for the top-level mission executor as well as all individual processes. Selection of a distributed multiprocessor architecture allows hardware-based fault tolerance, since failure of a given node can be functionally corrected by reloading and sharing the lost software modules on the remaining working processors.

Further work is needed to define formal specifications, characteristics and timing constraints for all NPS AUV software modules. Software module specifications need to include inputs and outputs, functionality, module dependencies, hard or soft scheduling constraints, periodic or aperiodic execution, relative priorities, expected frequency and duration, and all other parameters of importance to integrated system design.

The top priority for NPS AUV operating system software integration is to establish a new baseline architecture of system software running in the vehicle. This will allow more sophisticated operations and the addition of new processes to the basic control loop. It is unfortunate that most theses written about the NPS AUV to date have been unable to test their conclusions using the actual vehicle in the water.

Ensuring maximum processor utilization through improvements to the rate-monotonic scheduling algorithm is important work that is expected to continue by verifying current scheduling conclusions using actual NPS AUV processes. The incorporation of dynamic scheduling features holds great promise for the effective coordination of numerous distributed artificial intelligence software modules.

Perhaps the most interesting research immediately applicable to the NPS AUV is the investigation of alternate system software architecture organizations. Many possibilities are available which might incorporate multiple intelligent agents, low-level behaviors, expert systems and blackboard paradigms (Wright 88) (Durfee 88). A real-time architecture that allows flexible support of a variety of compatible software approaches will provide the best framework for rapid research progress.

The NPS AUV is a key project that integrates many of the critical technologies important to the Navy of tomorrow. The successful establishment of a reliable and robust real-time system software architecture will be the foundation that supports all future NPS AUV operations.

IX. PERFORMANCE EVALUATION AND FUTURE RESEARCH

A. SIMULATOR LIMITATIONS AND PERFORMANCE MEASUREMENTS

The NPS AUV Integrated Simulator is designed to eliminate as many design restrictions as possible in a distributed research environment. There are two primary limitations that restrict integrated simulator performance: graphics simulation program display rate and data transfer rates.

The NPS AUV Integrated Simulator graphics simulation program is written in ANSI C and uses the GL Graphics Library to run on a Silicon Graphics Inc. Iris/4D 240VGX workstation. The complexity of drawing panel interfaces and multiple complex objects restricts the playback speed the graphics simulation program can maintain. Maximum speed of playback with no associated environmental objects to be drawn is 7 Hz (i.e. 7 frames per second), nearly matching the telemetry sampling rate of 10 Hz. Worst case playback screen update rate using actual in-water test data has been 1-2 Hz. Playback rates are correspondingly lower on less capable Iris workstations. Although the worst case 1-2 Hz screen update rate may appear somewhat jerky to the user, screen vehicle motion accurately renders real-time vehicle motion since intermediate telemetry data records are skipped. Further improvements in the screen update rate are possible if control panel interface code is optimized and the overall simulation program is tuned and parallelized for peak performance.

Data transfer rates over the network currently do not impact integrated simulator performance since data packets and socket software are not yet implemented. However this can be a significant bottleneck that prevents realistic performance, given current experience with packet-passing simulation (Byrnes 92). Integrated simulator implementation of software sockets must be tolerant of packet delivery time delays or nondeterministic and incorrect results will occur.

B. INTEGRATED SIMULATOR FOLLOW-ON WORK

Several possibilities invite immediate follow-on work to the NPS AUV Integrated Simulator.

A vehicle hardware model and hydrodynamic vehicle response model need to be added to the simulator to provide realistic simulation of vehicle physical response using the laboratory AUV. Previous hydrodynamic response models have been part of the graphics simulation code. However network time delays do not allow accurate response or correct interaction between the laboratory AUV and the graphics workstation. Realistic integration of hydrodynamic response with simulation can be accomplished by making the model an independent OS-9 process or placing the model on a separate connected microprocessor inside the laboratory AUV backplane chassis. Validation of the model can be conducted through test comparison with actual in-water data.

Software sockets that allow passing data packets between processes need to be implemented in a way that is simple for any NPS AUV programmer to use. Socket implementations are already available for Unix processes, including GL-based graphics simulation programs. The key challenge will be to implement compatible software sockets for the NPS AUV source code running under the OS-9 operating system.

Sonar visualization capabilities have not been added to the graphics simulation program. Addition of graphics polygons to represent sonar beams, echo contact points and error boundaries will improve the user's ability to visualize real-time interactions between the vehicle and the environment.

C. POTENTIAL FUTURE RESEARCH

The NPS AUV Integrated Simulator provides a foundation for many types of future AUV research. The following areas are of particular interest.

The NPS AUV vehicle control software needs to allow modular addition of software processes shown in the block diagram of Figure 2.6. Current vehicle software is limited and supports only rudimentary behaviors. Access to the

laboratory AUV via the integrated simulator data network allows all interested researchers the opportunity to test their programs on the vehicle. The baseline AUV control software must be upgraded to support these increased demands.

Scientific visualization techniques hold great promise for rapid understanding of complex physical processes. Visualization can be used for comparison of theoretical and empirical data. Close evaluation of hydrodynamics and vehicle sideslip models may reveal general techniques for formal model verification.

Visualization of sonar and acoustic interactions is a promising area of research. Sonar visualization can be directly implemented in the integrated simulator for new and proposed sonar types. Sonar visualization will greatly increase user understanding of sonar performance and is likely to have tremendous tactical and training significance.

Simulation of world models is not a precisely defined science. Accurate simulation world models are needed for navigation, the external environment, hydrodynamic response, sonar acoustic behavior and physical vehicle hardware components. Additional research is needed to determine the specifications of these world models, validate their correctness and show how best to implement them in the context of an integrated simulator. Successful integration of general world models into a real-time simulator is a prerequisite to production of a virtual world where complete and realistic interaction is possible.

Numerous additional research examples are conceivable. Current and future researchers working on the NPS AUV project will undoubtedly develop their own applications and extensions using the NPS AUV Integrated Simulator.

X. SUMMARY

The development and testing of AUV hardware and software is greatly complicated by vehicle inaccessibility during operation. Integrated simulation remotely links vehicle components and support equipment with graphics simulation workstations. Integrated simulation allows complete real-time, pre-mission, pseudo-mission and post-mission visualization and analysis in the lab environment. Integrated simulator testing of software and hardware is a broad and versatile method that supports rapid and robust diagnosis and correction of system faults.

In order to fully understand the simulation requirements of demanding artificial intelligence processes necessary for AUV operation, in-depth studies are included for path planning, expert system sonar classification and real-time operating system considerations. Conclusions specific to these areas of research are included with each chapter.

High-resolution three-dimensional graphics workstations can provide real-time representations of vehicle dynamics, control system behavior, mission execution, sensor processing and object classification. The flexibility and versatility provided by this approach enables visualization and analysis of all aspects of AUV development. Integrated simulator networking is recommended as a fundamental requirement for AUV research and deployment. The availability of the NPS AUV Integrated Simulator for distributed research promises to benefit all future NPS AUV work.

APPENDIX A. NPS AUV INTEGRATED SIMULATOR USER'S GUIDE

The NPS AUV Integrated Simulator is designed to be accessible to anyone performing AUV-related work at NPS. This User's Guide shows how to utilize the integrated simulator for mission playback, software design or vehicle visualization. An additional example which shows how a standalone application can use the graphics simulation program to visualize NPS AUV behavior can be found in (Compton 92).

1. NPS AUV GRAPHICS SIMULATION EXECUTION

The graphics simulator program is executed by logging on one of the Iris workstations, changing to the appropriate directory (default *~brutzman/auv*) and typing *auvsim*. The *auvsim* graphics simulation program loads all high-level objects and commands in the current *pool.auv* file. Users can move their viewpoint and reference point around the simulated world, run telemetry replay files and reposition individual objects as desired.

2. NPS AUV INTEGRATED SIMULATOR CONTROL PANEL

A simple control panel has been provided as the user interface to the simulator. Two types of files may be entered in the filename type-in box: "filename.auv" high-level object files and "filename.d" telemetry files. High-level object files are used to enter objects into the simulated world, position the AUV, change the background environment graphics object and provide commands to the graphics simulation program. Telemetry replay files are files of floating point records, each of which represents NPS AUV state at a given time. File specifications are described in detail in Chapter III. Sample high-level object files and telemetry replay files are shown in Figures A.1 and A.2.

A great deal of functionality is included on the control panel. The name and number of the current object of interest is displayed along with position, scale, posture

```
; this is a comment for high level object file test.auv
; note free format and any comment allowed after data

clear
object   mine.off  40   20   0    scale 2   time 12
mine     70   43   1    1.5
circle   5    5    1    2
colors   255 0 0   1.0  0 0 255
;        line RGB-alpha-wall RGB
WALL     20   20   0    50   50   8
line     20   20   1    30   40   1
colors   255 255 0 1.0  0 255 0
;        line RGB-alpha-wall RGB
line     30   40   1    50   50   1
point    50   50   1
object   cylinder.off   15   15   1
cylinder 70   43   4    11
origin   20   20.1 1    this is the new reference
;                  origin for coordinates to follow
; the next point will be at 20 20.1 1
point    0    0    0
origin   0    0    0    origin reset to pool corner
AUV      20   10   2
environment   nps_pool.off   restores pool environment
gyroerror     -15.0 degrees
gyrodrift     -1.1  deg/min
replaysize    3    (type 3 = M35, 17 values per line)
replayfile    m35.d
depthband     0.0  8.0  all dimensions in feet
```

Figure A.1 Example high-level object file

```
0.000000 0.000000 0.000000 0.161064 -0.054569 0.050224 0.056758 -0.133328
    -0.017602 -0.031889 0.125950 -0.045393 0 0 313 1 1558
0.100000 0.199678 0.011346 0.158827 -0.078823 0.062779 0.057525 -0.155013
    -0.017602 -0.024220 0.131702 0.001013 0 0 308 1 1556
0.200000 0.399347 0.022844 0.149879 -0.072759 0.071150 0.057909 -0.192962
    -0.030343 -0.029333 0.130105 0.008301 0 0 303 1 1556
0.300000 0.599012 0.034419 0.147642 -0.057601 0.054409 0.057525 -0.133328
    -0.027795 -0.033168 0.127229 -0.041462 0 0 288 1 1554
0.400000 0.798681 0.045918 0.140931 -0.063664 0.058594 0.056375 -0.117064
    -0.031617 -0.034446 0.123713 -0.028987 0 0 285 1 1551
0.500000 0.998363 0.057187 0.138694 -0.042443 0.050224 0.055224 -0.100800
    -0.037988 -0.044672 0.115724 -0.071750 0 0 277 1 1551
0.600000 1.198058 0.068226 0.143168 -0.048506 0.048131 0.055224 -0.062850
    -0.037988 -0.037002 0.119559 -0.083995 0 0 280 1 1552
0.700000 1.397754 0.079265 0.147642 -0.063664 0.060687 0.055608 -0.095378
    -0.049454 -0.031889 0.123074 -0.073638 0 0 274 1 1555
0.800000 1.597444 0.090381 0.154353 -0.051538 0.056502 0.056758 -0.122485
    -0.058373 -0.034446 0.124672 -0.117965 0 0 276 1 1553
0.900000 1.797122 0.101727 0.161064 -0.051538 0.048131 0.058676 -0.127906
    -0.053277 -0.029333 0.132022 -0.141904 0 0 276 1 1559
1.000000 1.996778 0.113455 0.163301 -0.066696 0.050224 0.060210 -0.138749
    -0.040536 -0.017829 0.141609 -0.105188 0 0 268 1 1558
```

Figure A.2 One second excerpt of 10 Hz telemetry replay file

and time tag. Color objects can also be displayed. Color objects are a special object type which control the RGB (red green blue) values of geometric objects such as lines and walls which follow. Operator viewpoint and reference point control are provided through dials and sliders for *Height* between viewpoint and reference point, *Range* in the x-y direction, *Azimuth* between viewpoint and reference point in the x-y plane, and viewpoint *Twist* angle. With a little practice users can move with ease throughout the environment.

Control panel function buttons are also provided. The *Sonar* button toggles a plot display of sonar data versus time for left, right, forward and depth transducers. *Reset* returns the screen to its initial settings. *Snapshot* takes a black and white picture of the pool and allows the user to then select a small portion to be saved as a figure in

Silicon Graphics Inc. *rgb* format ("snapshot.rgb") and Encapsulated Postscript format ("snapshot.eps"). The *Snapshot* feature is very handy for creating figures to be imported by Frame or Wordperfect, but graphics files are very large so this feature should be used sparingly. More information on snapshot execution can be found in the shell script "screensnapshot". *Replay* executes or resumes the current telemetry replay file, and *Step* single steps through the telemetry replay file one record at a time. Telemetry data may be viewed by selecting the AUV object or popping up the Unix command line window over the screen display. *Exit* quits the simulation program. Buttons are selected by positioning the mouse cursor and holding down the left mouse button. Button response is indicated by a button color change.

The right mouse button selects a multiple level menu. Menu control is provided for viewpoint, reference point, object position, rotation and scaling, and special simulator features such as lighting model, real-time playback toggle and system usage performance meter.

The dials and button box are also operative. Buttons 1-2-3 and 5-6-7 move viewpoint x-y-z coordinates towards or away from the reference point respectively. Button 4 is *Reset* and button 8 is *Exit*. Dial 6 selects the next or previous object. Dials 4-2-0 pan viewpoint and reference point together in x-y-z directions respectively. Dials 7-5-3-1 are *Height*, *Range*, *Azimuth* and *Twist* respectively. Dial and button physical configurations are shown in Figure A.3.

3. LABORATORY GESPAC EXECUTION

NPS AUV control loop software can be compiled and executed on network node *auvsim1*, the laboratory Gespac version of the NPS AUV microprocessor. Figures A.4 through A.8 show a sample logon, compilation and execution of NPS AUV control software program *loop.c*. Following execution the resulting telemetry replay file "d.d" is transferred to a graphics workstation where it can be replayed by the graphics simulation program *auvsim*.

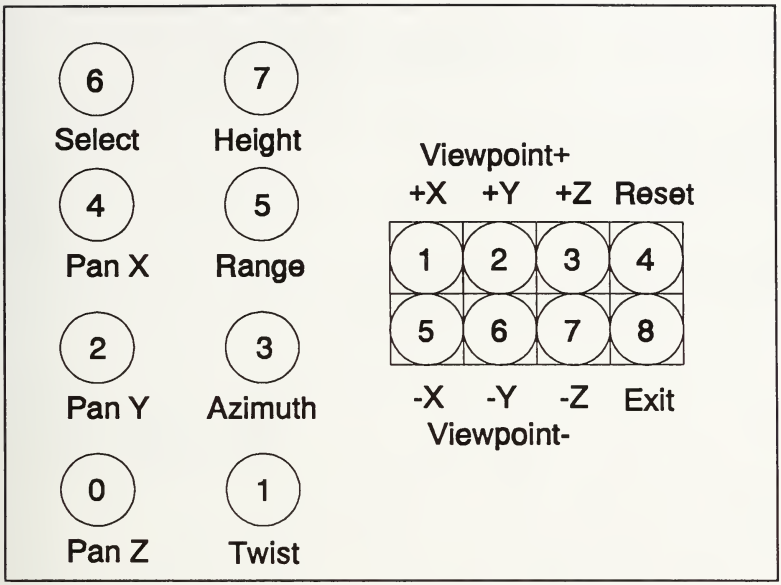


Figure A.3 NPS AUV Integrated Simulator dials and buttons

```
File auvsim.log:        sample execution of AUV software on
                        Iris Lab AUV Gespac running OS-9 on 9 JAN 91

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

gemini:/n/gemini/work/brutzman>> telnet auvsim1
Trying 131.120.1.40 ...
Connected to auvsim1.cs.nps.navy.mil.
Escape character is '^]'.

OS-9/68K V2.3   Gescomp 84xx/86xx - 68020   92/01/09 00:32:05

User name?: brutzman
Password:
Process #06 logged on    92/01/09 00:32:13
Welcome!

  ******   WELCOME TO PROFESSIONAL OS-9/68k V2.3   ******
   *                                                    *
    *   S   C     /H0    =  Hard Disk #1  (20 Mb)       *
     *       O                                          *
      *   Y   N     /D0    =  Floppy Disk #1  (FDC-3)     *
       *       F                                        *
        *   S   I                                       *
         *       G                                      *
          *   T   U     /TERM  =  RS-232 Interface to Terminal  *
           *       R                                    *
 auvsim1    *   E   A     /T1    =  Auxiliary Serial Port #1     *
             *       T                                  *
 NPS AUV      *   M   I    [/P     =  Centronics-type Printer]     *
               *       O                                *
 Integrated     *       N                               *
                 *                                      *
 Simulator        ********************************************************


$ dir
                          Directory of . 00:32:22
ARC             AUV             AUVRUN          AUV_OLD         BYRNES
C               CEE             CMDS            DEFS            ETC
INET            INSTALL         LIB             MACROS          MISC
MIT             MSDOS           OS9boot         Read_Me
Release.notes
SUBMODS         SYS             SYSSRC          TEMP            WL
d.d             dir.fix         ev0.txt         hello.c         junk
loop            loop.c          lp.d            macph           mkrom.bat
readme.utils    romlist.sho     startlan        startlan.ori    startup
startup.bak     startup.cc      startup.env     startup.full    startup.ng
startup.ok      startup.old     startup.ori     t2.19200        t2.9600
```

Figure A.4 Script of laboratory GESPAC execution of NPS AUV control loop software (part 1)

```
$ * comments start with an asterisk

$ * loop.c is the closed-loop control software used in the AUV

$ * lp.d is the run geometry data file

$ cc loop.c -k2f

'loop.c'
cpp:
c68:
o68:
r68:
l68:

$ * Now execute loop.  Initial values were values read from lp.d data file.

$ * start_dwell is the only value that matters right now since it is the

$ *    initial delay time.

$ loop
10.000000 0.000000 2.000000
30.000000 0.000000 2.000000
55.000000 3.140000 2.000000
80.000000 3.140000 2.000000
100.000000 6.280000 2.000000
Input start_dwell
1
Input  k_psi and k_r
2.5
.5
Input k_z, k_theta, and k_q
-1.1
3.5
2.5
Input k_speed and ki_speed
4.0
0.5
Input speed_limit from 1.0 to 3.0 feet/sec
1.5
Input rpm from +-200.0 to +-650.0, type 400.0
400.0
  Position AUV for Directional Gyro Offset Measurement
  Rate Gyro zero measurement
  Hit Any Key When Ready

pitch_0 = 0
roll_0 = 0
roll_rate_0 = 0
pitch_rate_0 = 0
yaw_rate_0 = 0
z_val0 = 0
dg_offset = 0.097793
Starting
Error #000:002  keyboard quit

$ * program exit was accomplished using control-E after about 20 seconds
```

Figure A.5 Script of laboratory GESPAC execution of NPS AUV control loop software (part 2)

```
$ * AUV output file is d.d and contains

$ *     the usual floating point state vector at 10 Hz.

$ *     First parameter is clock time.

$ list d.d
0.000000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 576.450000
-8.479200 -8.479200
0.100000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 603.900000
-8.479200 -8.479200
0.200000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 631.350000
-8.479200 -8.479200
0.300000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.400000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.500000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.600000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.700000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.800000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
0.900000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.000000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.100000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.200000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.300000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.400000 0.000000 0.000000 0.000000 0.000000 -0.000000 0.000000
0.000000 0.000000 -0.000000 2.200000 0.000000 0.000000 0.000000 650.000000
-8.479200 -8.479200
1.500000 0.000000 0.000000 0.00
$ Read I/O error - Error #000:002  keyboard quit

$ * Control-E again used to break out
```

Figure A.6 Script of laboratory GESPAC execution of NPS AUV control loop software (part 3)

```
$ * Now let's send d.d to the iris for use by the integrated simulator

$ ftp iris1
Connected to gravy1.cs.nps.navy.mil.
220 gravy1 FTP server (IRIX version 5.46 Aug  6 1990 10:17) ready.
Name (gravy1.cs.nps.navy.mil:brutzman):
Password (gravy1.cs.nps.navy.mil:brutzman):
331 Password required for brutzman.
230 User brutzman logged in.
Connected to gravy1.cs.nps.navy.mil.
Mode: stream       Type: ascii        Form: non-print    Structure: file
Verbose: on        Bell: off          Prompting: on      Globbing: on
Hash mark printing: off               Use of PORT commands: on
ftp> put d.d labtest.d
200 PORT command successful.
150 Opening ASCII mode data connection for 'labtest.d'.
226 Transfer complete.
2575 bytes sent in 0.10 seconds (25.15 Kbytes/s)
ftp> quit
221 Goodbye.

$ telnet iris1
Trying 131.120.1.20...Connected to gravy1.cs.nps.navy.mil.
Escape character is '^]'.
capture closed.


IRIX System V.3 (gravy1)

login: brutzman
Password:
IRIX System V Release 3.3.1 gravy1
Copyright (c) 1988,1989,1990 Silicon Graphics, Inc.
All Rights Reserved.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        SCHEDULED DOWN TIME
                   Backups --  Wednesdays 0800-0900
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

gravy1:/n/gravy1/work/brutzman
% ls
AUV             backup.pool     graphics        nps             tape.c
GA              bay_20m_data    lab.d           off             temp
WorkSpace       clips           labtest.d       pline           titler
auto_pilot.c    d.d             laser           pline.c
auv             dumpster        laser2          pool_m35.auv
auvIII          ff              laser2h         preview
auv_long_term   g               laserh          robotics

gravy1:/n/gravy1/work/brutzman
% logout
Connection closed by foreign host.
```

Figure A.7 Script of laboratory GESPAC execution of NPS AUV control loop software (part 4)

```
$ * Note file d.d was successfully transferred as file labtest.d to the
iris

$ * also note ftp and telnet can be used back and forth as well as nested.

$ * Lack of analog/digital cards did not prevent loop.c from continuing.

$ * This script file is saved as auvsim.log


$ * Due to a shell glitch, two 'logout' commands are needed to exit OS-9.

$ logout
brutzman> logout
Connection closed by foreign host.
gemini:/n/gemini/work/brutzman>>
```

Figure A.8 Script of laboratory GESPAC execution of NPS AUV control loop software (part 5)

APPENDIX B. NPS AUV GRAPHICS SIMULATION PROGRAM SYNOPSIS

The NPS AUV Integrated Simulator graphic simulation program is written in ANSI C to run on Silicon Graphics Inc. Iris workstations. Numerous calls are made to GL Graphics Library routines. Platform dependence and length (5200 lines of code) makes complete reproduction of the source program and support files impractical. Source programs are summarized in this appendix and are available via Internet as described in Appendix F.

1. GRAPHICS SIMULATION PROGRAM STRUCTURE

The graphics simulation program has a structure typical of most Iris applications. Graphics functions are initialized, data is initialized and a world view is drawn from the default viewpoint. A main graphics loop is then repeated indefinitely which reads user inputs (if any), redraws the world view and outputs updated values. Screen update rate is dependent on workstation capabilities and the changing complexity of the world view being drawn. Figure B.1 shows the basic simulation graphics loop and summarizes all major functions.

2. NPS PANEL DESIGNER

The NPS Panel Designer (NPSPD) is used to provide a user-friendly control panel interface (King Prevatt 90). NPSPD reads high-level text files that specify layout and functionality of a wide variety of interface icons such as meters, dials, knobs, buttons etc. NPSPD then automatically generates "C" source code functions that can be tied to the target application through *Makefile* entries and *define* statements. Although initial setup is difficult, use of NPSPD permits extensive changes to the user interface. Subsequent modifications using NPSPD are rapid and convenient, permitting flexible development of the user interface during simulator

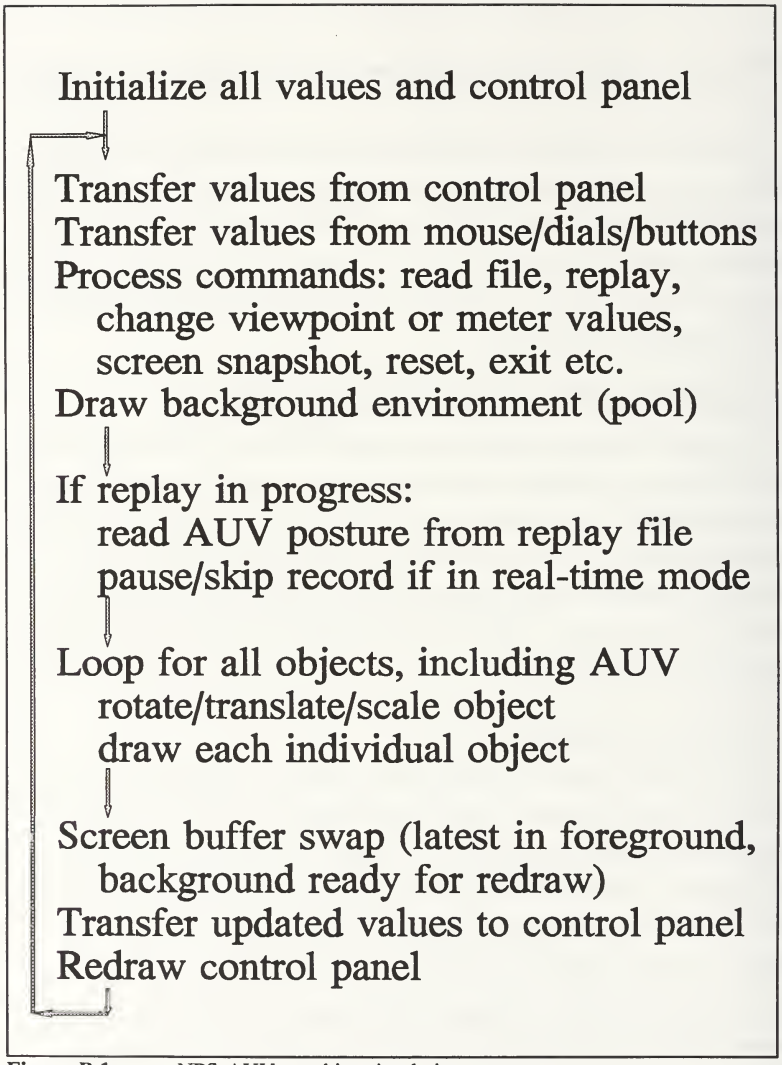


Figure B.1 NPS AUV graphics simulation program

development. NPSPD specifications and examples are found in (King Prevatt 90) and (Jurewicz 90).

3. GRAPHIC OBJECT MODELING USING OBJECT FILE FORMAT (OFF)

Creation and modeling of complex graphic objects such as underwater mines or vehicles can be extremely tedious if only using GL Graphics Library geometric primitive function calls. Object File Format (OFF) has been developed at NPS to encapsulate most of the functionality of GL Graphics Library lighting, drawing and texturing function calls in a manner that supports modeling and manipulation of high-level graphics objects (Zyda 91). Use of OFF objects greatly simplifies treatment of individual objects modeled in a simulated world.

APPENDIX C. NPS AUV SONAR CLASSIFICATION SYSTEM SOURCE CODE

```
;_____________________________________________________________________________
;
;                AUV Sonar Expert System
;
;   Filename:    auvsonar
;
;   Purpose:     Batch file for auvsonar.clp which resets and executes the
;                AUV sonar contact classification expert system.
;
;   Paper:       "Autonomous Underwater Vehicle Sonar Classification using
;                 Expert Systems and Neural Networks"
;
;                IEEE OCEANS '92 Conference, Newport, Rhode Island
;
;   Authors:     Don Brutzman, Mark Compton and Dr. Yutaka Kanayama
;
;   Date:        24 November 91
;
;   Execution:   unix>  clips5                    unix>  clips5
;                CLIPS> (load auvsonar.clp)       CLIPS> (batch auvsonar)
;                CLIPS> (reset)                   CLIPS> (run)
;                CLIPS> (run)
;
;_____________________________________________________________________________

; Clear & close files in case they were left open during previous execution

       (clear)              ; clear all facts and rules
(close rangefile)     ; Close AUV-recorded pool test data input file
(close plotfile)      ; Close xy coordinate file used for graph output
(close auvfile)       ; Close expert system classification output file

;_____________________________________________________________________________


       (load auvsonar.clp)  ; Load in AUV Sonar Classification Expert System

       (undefrule oldarea1)
       (undefrule oldarea2)

       (reset)              ; Initialize agenda and assert initial facts

;;;;;  (run)                ; Execute AUV Sonar Classification Expert System
```

```
;_____________________________________________________________________________
;
;               AUV Sonar Expert System
;
;   Filename:   auvsonar.clp
;
;   Purpose:    Define data templates, rules, functions and user interface
;               for the AUV sonar contact classification expert system.
;
;   Paper:      "Autonomous Sonar Classification using Expert Systems"
;               IEEE Oceanic Engineering Society
;               IEEE OCEANS '92 Conference, Newport, Rhode Island
;
;   Authors:    Don Brutzman and Mark Compton
;   Advisor:    Dr. Yutaka Kanayama
;
;   Date:       1 March 91
;
;   Comments:   This expert system takes data files generated by the NPS AUV,
;               uses sonar returns and AUV position to generate locations of
;               sonar contacts, perform two-dimensional linear regression to
;
;               build line segments, combine segments into polyhedrons and
;               then determines the probable classification of each polyhedron.
;
;   Language:   CLIPS "C" Language Integrated Production System
;
;   Execution:   unix>  clips5              |    unix>  clips5
;               CLIPS> (load auvsonar.clp)  |   CLIPS> (batch auvsonar)
;               CLIPS> (reset)              |   CLIPS> (run)
;               CLIPS> (run)
;
;               Execution 'dribble' files are saved in auvsonar.log
;
;   References: _Sonar Data Interpretation for Autonomous Mobile Robots_,
;               Yutaka Kanayama, Tetsuo Noguchi, and Bruce Hartman,
;               unpublished paper.
;
;   History:    Original program development for CS4311 Expert Systems
;               taught by Dr. Kanayama.
;
;   Caveat:     The NPS pool coordinate system is the world reference used
;               where x is pool length, y is pool width, and z is pool depth.
;
;   Status:     Initial development complete for object classification.
;               Full pool depth used for pool object outputs.
;               Initial offset option for centering pool data included.
;               Verbose output option and excess data retraction completed.
;               Gyro error/gyro drift rate evaluation & correction implemented.
;               Centroid and cross-sectional area calculations done for objects.
;               Top-level classification of objects using area is possible.
;               Mine classification implemented satisfactorily.
;               Excessively narrow objects are reclassified as walls.
;
;_____________________________________________________________________________


;_____________________________________________________________________________
;
;   Data Type Deftemplates
;_____________________________________________________________________________
;
;   Data template and slot names correspond to AUV Data Dictionary definitions.
;   Data template names have their first letter capitalized.
;   Variable names are all lower case.
;   CLIPS data types and symbols used in symbolic slots are capitalized.
;
;_____________________________________________________________________________

(deftemplate Range_data

  (field time                           ; time is positive, set by AUV
               (type    NUMBER)
               (default 0)              ; time zero is used for dummy facts
               (range   0 ?VARIABLE))
  (field x                              ; element of Point_3D AUV data type
               (type    NUMBER)        ; dead reckoning estimate of travel
               (default 0)             ;    relative to start position
               (range   0 ?VARIABLE))
  (field y                              ; element of Point_3D AUV data type
```

```
                    (type    NUMBER)        ; dead reckoning estimate of travel
                    (default 0)             ;    relative to start position
                    (range   0 ?VARIABLE))
  (field z                                  ; element of Point_3D AUV data type
                    (type    NUMBER)        ; source:  pressure-sensing depth cell
                    (default 0)             ; which may be inaccurate when shallow
                    (range   0 ?VARIABLE))
  (field phi                                ; element of Attitude_3D AUV data type
                    (type    NUMBER)        ;   in radians
                    (default 0))            ;    (roll)
  (field theta                              ; element of Attitude_3D AUV data type
                    (type    NUMBER)        ;   in radians
                    (default 0))            ;    (pitch)
  (field psi                                ; element of Attitude_3D AUV data type
                    (type    NUMBER)        ;   in radians.  Note caveat on pg. 1
                    (default 0))            ;    (yaw)
  (field p                                  ; element of Point_3D AUV data type
                    (type    NUMBER)        ;   in radians/sec
                    (default 0)
                    (range   0 ?VARIABLE))
  (field q                                  ; element of Point_3D AUV data type
                    (type    NUMBER)        ;   in radians/sec
                    (default 0)
                    (range   0 ?VARIABLE))
  (field r                                  ; element of Point_3D AUV data type
                    (type    NUMBER)        ;   in radians/sec
                    (default 0)
                    (range   0 ?VARIABLE))
  (field delta_dive_planes                  ; change in bow/stern planes position
                    (type    NUMBER)        ;   in degrees
                    (default 0)
                    (range   0 ?VARIABLE))
  (field delta_rudders                      ; change in rudder planes position
                    (type    NUMBER)        ;   in degrees
                    (default 0)
                    (range   0 ?VARIABLE))
  (field range_a                            ; 0-4095 range units correspond to
                    (type    NUMBER)        ;   0..30m pool or 0..300m ocean.
                    (default 0)
                    (range   0 ?VARIABLE))
  (field range_b                            ; Up to 4 transducers can be included.
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field range_c
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field range_d
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field valid_a                            ; Validity signal from sonar hardware
                    (type    INTEGER)
                    (default 1))
  (field valid_b
                    (type    INTEGER)
                    (default 1))
  (field valid_c
                    (type    INTEGER)
                    (default 1))
  (field valid_d
                    (type    INTEGER)
                    (default 1))
  (field speed                              ; AUV speed from flow sensor
                    (type    NUMBER)
                    (default 0.0))
  (field processed                          ; set TRUE when point is asserted,
                    (type    SYMBOL)        ;     FALSE until then.
                    (default FALSE)
            (allowed-values TRUE FALSE))
)

;_____________________________________________________________________________

(deftemplate Object_data

  (field detection_time                     ; time is positive, set by AUV
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field latest_time
```

```
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field valid
                    (type    INTEGER)
                    (default 0))
  (field x                                  ; object center
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field y                                  ; object center
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field z                                  ; object center
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field accuracy
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 ?VARIABLE))
  (field object
                    (type    INTEGER)
                    (default 0)
                    (range   0 9))
  (field length
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field height
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field width
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field confidence                         ; normalized
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 1.0))
)

;_____________________________________________________________________________

(deftemplate Point

  (field time                               ; time is positive, set by AUV
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field x                                  ; element of Point_3D AUV data type
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field y                                  ; element of Point_3D AUV data type
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field z                                  ; element of Point_3D AUV data type
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field valid
                    (type    INTEGER)
                    (default 0))
  (field status
                    (type    SYMBOL)
                    (default NEW)
            (allowed-values NEW ACTIVE INVALID ENDPOINT USED))
)

;_____________________________________________________________________________

(deftemplate Regression_line

  (field start                                  ; matches time of start point
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field end                                    ; matches time of end point
```

```
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field r
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 ?VARIABLE))
  (field orientation                              ; normalized degrees
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 ?VARIABLE))
  (field correlation
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 ?VARIABLE))
  (field status
                    (type    SYMBOL)
                    (default NEW)
            (allowed-values NEW CURRENT VALID USED USED_FOR_AREA))
)

;_____________________________________________________________________________

(deftemplate Node

  (field time
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field x
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field y
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field z
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field accuracy
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 ?VARIABLE))
  (field confidence
                    (type    FLOAT)
                    (default 0.0)
                    (range   0.0 1.0))
)

;_____________________________________________________________________________

(deftemplate Edge

  (field start                    ; slot values are times corresponding to data
                    (type    FLOAT))
  (field end
                    (type    FLOAT))
  (field averagez
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field status
                    (type    SYMBOL)
                    (default USED)
            (allowed-values USED USED_FOR_AREA))
)

;_____________________________________________________________________________

(deftemplate Curve                        ; not yet implemented

  (field time
                    (type    NUMBER)
                    (default 0)
                    (range   0 ?VARIABLE))
  (field node                     ; slot values are times corresponding to data
                    (type    FLOAT))
  (field edge
                    (type    FLOAT))
  (field shape
```

```
                (type    SYMBOL))
)

;_____________________________________________________________________________

(deftemplate Polyhedron

  (field start                    ; time of the initial node/edge/curve element
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field end                      ; time of most recent node/edge/curve element
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field startx
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field starty
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field startz
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field centroidx
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field centroidy
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field centroidz
                (type    NUMBER)
                (default 0)
                (range   0 ?VARIABLE))
  (field sidecount
                (type    INTEGER)
                (default 1)
                (range   1 ?VARIABLE))
  (field sidecounter1
                (type    INTEGER)
                (default -1)
                (range   -1 ?VARIABLE))
  (field sidecounter2
                (type    INTEGER)
                (default -1)
                (range   -1 ?VARIABLE))
  (field area
                (type    FLOAT)
                (default 0.0)
                (range   0.0 ?VARIABLE))
  (field height
                (type    FLOAT)
                (default 0.0)
                (range   0.0 ?VARIABLE))
  (field accuracy
                (type    FLOAT)
                (default 0.0)
                (range   0.0 ?VARIABLE))
  (field confidence
                (type    FLOAT)
                (default 0.0)
                (range   0.0 1.0))
  (field trait
                (type    SYMBOL))
  (field status
                (type    SYMBOL)
                (default ACTIVE)
           (allowed-values ACTIVE COMPLETE USED_FOR_AREA))
  (field classification
                (type    SYMBOL)
                (default WALL)
           (allowed-values NEW CURRENT WALL OBJECT MINE SWIMMER UNKNOWN
                           SEA-MOUNT SUBMARINE SHIP_I_E_SKIMMER_PUKE
                           BIOLOGICS LOTS_OF_BIOLOGICS))
)

;_____________________________________________________________________________
```

```
;
; Initialization of Flag Facts
;_____________________________________________________________________

(deffacts initial-flags

        (start-new-window-flag)         ; commence parametric regression process

        (retract-excess-data    TRUE)   ; any other value saves excess data

        (location               pool)   ; NPS swimming pool test environment

;;;;;;;  (location              ocean)  ; alternative environment

)

;_____________________________________________________________________
;
; Global Constants
;_____________________________________________________________________

(defglobal ?*minimum_points_in_edge* = 5) ; hard coded into regression defrules

(defglobal ?*transducer_a* = 1); forward transducer corresponds to slot range_a
(defglobal ?*transducer_b* = 2); left    transducer corresponds to slot range_b
(defglobal ?*transducer_c* = 3); right   transducer corresponds to slot range_c
(defglobal ?*transducer_d* = 4); depth   transducer corresponds to slot range_d

(defglobal ?*feet_per_sonar_unit* = 0.02398) ; 0-4095 range units correspond to
                                             ;   0..30m pool or 0..300m ocean.

(defglobal ?*correlation_confidence_weight* = 1)
(defglobal ?*validity_confidence_weight*    = 1)

(defglobal ?*cl*        = 3.00)     ; max # standard deviations a point can be out
(defglobal ?*c2*        = 2.00)     ; max offset distance (feet) allowed from line
(defglobal ?*c3*        = 0.066)    ; min regression ellipse thinness requirement

; define default line/wall color strings
(defglobal ?*color1* = "Color       0  255  78 1.0   0 255  78"); detected edge
(defglobal ?*color2* = "Color     200  255 150 0.7 200 255 150"); inferred edge
(defglobal ?*color3* = "Color       0   78 255 0.5   0  78 255");  hidden  edge

(defglobal ?*min_wall_length*   = 1.0)   ; min allowable individual edge length
                                         ; to be output as a WALL

(defglobal ?*max_edge_distance* =  7.0)  ; max allowable distance between edges
                                         ; for edge-joining/polyhedron building

(defglobal ?*max_edge_angle*    = 10.0)  ; max allowable angle between edges
                                         ; for edge-joining/WALL building

(defglobal ?*wall_thinness_ratio* = 0.1) ; used to reclassify long skinny object
                                         ; as WALL

;_____________________________________________________________________
;
; Global Variables
;_____________________________________________________________________

(defglobal ?*n*         = 0.0)      ; m 00
(defglobal ?*sumx*      = 0.0)      ; m 10
(defglobal ?*sumy*      = 0.0)      ; m 01
(defglobal ?*sumxy*     = 0.0)      ; m 11
(defglobal ?*sumxx*     = 0.0)      ; m 20
(defglobal ?*sumyy*     = 0.0)      ; m 02

(defglobal ?*meanx*     = 0.0)      ; mu x
(defglobal ?*meany*     = 0.0)      ; mu y

(defglobal ?*sigmaxx*   = 0.0)      ; M 20
(defglobal ?*sigmaxy*   = 0.0)      ; M 11
(defglobal ?*sigmayy*   = 0.0)      ; M 02

(defglobal ?*phi*       = 0.0)      ; regression line orientation
(defglobal ?*r*         = 0.0)      ; regression line distance from origin

(defglobal ?*M-major*   = 0.0)      ; Moment around ellipse major axis
(defglobal ?*M-minor*   = 0.0)      ; Moment around ellipse minor axis

(defglobal ?*d-major*   = 0.0)      ; diameter on ellipse major axis
(defglobal ?*d-minor*   = 0.0)      ; diameter on ellipse minor axis
```

```
(defglobal ?*rho*       = 0.0)      ; ratio of major to minor axis diameters
(defglobal ?*delta*     = 0.0)      ; residual of a point
(defglobal ?*sigma*     = 0.0)      ; standard deviation

(defglobal ?*projection-x* = 0.0)  ; x projection of point i on major axis
(defglobal ?*projection-y* = 0.0)  ; y projection of point i on major axis

(defglobal ?*minx*      = 145)      ; for plot/graph boundaries
(defglobal ?*maxx*      = -15)      ;

(defglobal ?*miny*      = -50)      ; for plot/graph boundaries
(defglobal ?*maxy*      = 110)      ;

(defglobal ?*minz*      = 0.0)      ; for the currently active edge only
(defglobal ?*maxz*      = 0.0)      ; for the currently active edge only

(defglobal ?*defaultz* = 2.0)       ; default pool depth to be used for objects
                                    ;   of unspecified or indeterminate depth

(defglobal ?*offsetx*   = 0.0)      ; displacement added to (x, y, z) positional
(defglobal ?*offsety*   = 0.0)      ;    data to account for distance of the AUV
(defglobal ?*offsetz*   = 0.0)      ;    from the origin (i.e. corner) of the
                                    ;    NPS pool coordinate system

(defglobal ?*time*      = 0.0)      ; used to measure execution time

(defglobal ?*out*       = stdout)   ; verbose output default to stdout
                                    ;   otherwise ?*out* is reset to nil


(defglobal ?*gyroerror*          = 0.0)  ; User-provided gyro error (degrees)
(defglobal ?*newgyroerror*       = 0.0)  ; Expert system gyro error (degrees)
(defglobal ?*gyroerrortime*      = 0.0)  ; Average time of first wall found, used
                                         ;  as input to drift rate computation

(defglobal ?*gyrodriftrate*      = 0.0)  ; User-provided drift rate
(defglobal ?*newgyrodriftrate*   = 0.0)  ; Expert system drift rate

(defglobal ?*number_of_fields*   = 17)   ; read only actual # of input fields


;_____________________________________________________________________________

; A sample fact (for syntax training use only!)

(deffacts rangel (Range_data (time 0)
                             (x   2)  (y     3) (z    4)
                             (phi 5)  (theta 6) (psi  7)
                             (p   8)  (q     9) (r   10)
                             (delta_dive_planes      11)
                             (delta_rudders          12)
                             (range_a 13)    (valid_a 1)
                             (range_b 15)    (valid_b 1)
                             (speed 17) (processed TRUE))
)

;_____________________________________________________________________________
;
; Functions
;_____________________________________________________________________________

;  atan2 function matches C language and class text syntax.
;  Calling order:  (atan2 y x)

(defmethod atan2 ((?y NUMBER)(?x NUMBER (> ?x 0)))
  (atan (/ ?y ?x)))

(defmethod atan2 ((?y NUMBER (> ?y 0)) (?x NUMBER (< ?x 0)))
  (+ (atan (/ ?y ?x)) (pi)))

(defmethod atan2 ((?y NUMBER (< ?y 0)) (?x NUMBER (< ?x 0)))
  (- (atan (/ ?y ?x)) (pi)))

(defmethod atan2 ((?y NUMBER (> ?y 0)) (?x NUMBER (= ?x 0)))
  (/ (pi) 2.0))

(defmethod atan2 ((?y NUMBER (< ?y 0)) (?x NUMBER (= ?x 0)))
  (/ (pi) -2.0))

(defmethod atan2 ((?y NUMBER (= ?y 0)) (?x NUMBER (< ?x 0)))
  (pi))
```

```
(defmethod atan2 ((?y NUMBER (= ?y 0)) (?x NUMBER (= ?x 0)))
  0.0)

;_____________________________________________________________________________

(deffunction normalize (?x)  ; x in degrees, resulting range (0 .. 360)

   (bind ?norm ?x)
   (while (<  ?norm 0.0)    (bind ?norm (+ ?norm 360.0)))
   (while (>= ?norm 360.0) (bind ?norm (- ?norm 360.0)))
   ?norm
)

;_____________________________________________________________________________

(deffunction normalize2 (?x)  ; x in degrees, resulting range (-180 .. 180)

   (bind ?norm ?x)
   (while (<  ?norm -180.0) (bind ?norm (+ ?norm 360.0)))
   (while (>= ?norm  180.0) (bind ?norm (- ?norm 360.0)))
   ?norm
)

;_____________________________________________________________________________

(deffunction avg (?number1 ?number2)
   (/ (+ ?number1 ?number2) 2.0))

;_____________________________________________________________________________

(deffunction degrees (?x)  ; x in radians

   (/ (* ?x 180.0) (pi))
)

;_____________________________________________________________________________

(deffunction radians (?x)  ; x in radians

   (* (/ ?x 180.0) (pi))
)

;_____________________________________________________________________________

; Boolean function to ask a yes/no question

(deffunction yes-or-no (?question)        ; '?question' is the question string

  (format t "%n%s?  " ?question) ; ask the question
  (bind ?answer (lowcase (sym-cat (read))))

  (while (and (neq ?answer yes)  (neq ?answer y)  (neq ?answer yep)
              (neq ?answer yeah) (neq ?answer ye) (neq ?answer yea)
              (neq ?answer no)   (neq ?answer n)
              (neq ?answer nope) (neq ?answer nah))
         (format t "%n  Please answer yes or no:  ")
         (bind ?answer (lowcase (sym-cat (read)))))

  (if (or (eq ?answer yes)  (eq ?answer y)  (eq ?answer yep)
          (eq ?answer yeah) (eq ?answer ye) (eq ?answer yea))
      then TRUE
      else FALSE)
)

;_____________________________________________________________________________

(deffunction distance (?x1 ?y1 ?z1 ?x2 ?y2 ?z2)

   (sqrt (+ (* (- ?x1 ?x2) (- ?x1 ?x2))
            (* (- ?y1 ?y2) (- ?y1 ?y2))
            (* (- ?z1 ?z2) (- ?z1 ?z2))
    ))
)

;_____________________________________________________________________________
;
; Triangle S and area calculation functions

;_____________________________________________________________________________
;
(deffunction S    (?node1x ?node1y ?node2x ?node2y ?node3x ?node3y)
```

```
;   CCW triples are positive & CW triples are negative, matching conventions.

    (bind ?trianglearea
                        (* 0.5 (- (* (- ?node2x ?node1x) (- ?node3y ?node1y))
                                  (* (- ?node3x ?node1x) (- ?node2y ?node1y)))))
?trianglearea)

;_____________________________________________________________________________
;
(deffunction area (?node1x ?node1y ?node2x ?node2y ?node3x ?node3y)

;   area values are always positive, matching conventions.

    (bind ?trianglearea
                    (abs (* 0.5 (- (* (- ?node2x ?node1x) (- ?node3y ?node1y))
                                   (* (- ?node3x ?node1x) (- ?node2y ?node1y))))))
?trianglearea)

;_____________________________________________________________________________
;
; Expert system start and data file reading rules
;_____________________________________________________________________________

(defrule get-initial-expert-system-parameters-and-open-range-file

  (declare (salience 100))
  (initial-fact)
=>
  (dribble-off)
  (system "mv -f auvsonar.log auvsonar.log.bak")
  (dribble-on auvsonar.log)

  (printout t crlf crlf "Name of range data file to open?  ")
  (bind ?filename (read))
  (open  ?filename rangefile "r")
  (printout t "Opened range data file " ?filename  crlf)

  (printout t crlf)
  (if   (yes-or-no "Are there more than 17 fields per Range_data record")
   then (printout t crlf "Enter number of data fields per record:  ")
        (bind ?*number_of_fields* (read))
        (while (or (< ?*number_of_fields* 17) (> ?*number_of_fields* 20))
               (printout t crlf "Enter a value from 17..20:  ")
               (bind ?*number_of_fields* (read))
               (printout t crlf crlf)))

  ;     Determine output device for trace statements using 'format ?*out*'
  (printout t crlf)
  (if   (yes-or-no "Do you want verbose output onscreen during analysis")
   then (bind ?*out* stdout)
   else (bind ?*out* nil))

  (printout t crlf)
  (if   (yes-or-no "Do you want to input gyro error and gyro drift rate")
   then (printout t crlf "Enter gyro error      (degrees):  ")
        (bind ?*gyroerror* (normalize2 (read)))
        (printout t crlf "Enter gyro drift rate (degs/hr):  ")
        (bind ?*gyrodriftrate* (read)))
  (printout t crlf)

  (printout t crlf)
  (printout t crlf "Enter offset distance to be added to X positions to ")
  (printout t "account for the initial AUV displacement from pool corner:   ")
  (bind ?*offsetx* (read))
  (printout t crlf)

  (printout t crlf "Enter offset distance to be added to Y positions to ")
  (printout t "account for the initial AUV displacement from pool corner:   ")
  (bind ?*offsety* (read))
  (printout t crlf)

  (printout t crlf "Enter offset depth    to be added to Z positions to ")
  (printout t "account for the initial AUV displacement from pool surface:  ")
  (bind ?*offsetz* (read))
  (printout t crlf)

  (printout t crlf "Saving previous files pool.graph and pool.auv:" crlf)
  (printout t "mv -f  pool.auv    pool.auv.bak")
  (system     "mv -f  pool.auv    pool.auv.bak")
  (printout t crlf)
  (printout t "mv -f  pool.graph  pool.graph.bak")
  (system     "mv -f  pool.graph  pool.graph.bak")
```

```
  (printout t crlf)

  (open  "pool.auv"   auvfile  "a")
  (open  "pool.graph" plotfile "a")

  (printout auvfile crlf crlf
                    "        NPS AUV Sonar Classification Expert System"
                    "  (pool data " ?filename ")"
                    crlf crlf crlf)
  (printout auvfile crlf "All data values & type specifications are "
                         "defined by the AUV Data Dictionary."
                    crlf)
  (printout auvfile crlf "All coordinate values are relative to the "
                         "NPS Pool Coordinate System."
                    crlf crlf crlf)
  (printout auvfile crlf "AUV                "
                         ?*offsetx* "   " ?*offsety* "   " ?*offsetz*
            "  (xyz distances from AUV start position to pool origin)" crlf)

  (printout plotfile " 105.0  95.0 " crlf "\"NPS AUV Sonar Classification "
                    "Expert System  (pool data " ?filename ") \" "
                    crlf)

  (printout plotfile " 100.0 -30.0 " crlf "\"AUV start..origin offset values:  "
                                          ?*offsetx* ", "
                                          ?*offsety* ", "
                                          ?*offsetz*         " \" "
                    crlf)

  (printout plotfile " 105.0 -40.0 " crlf "\"Parametric regression constants:  "
                                            "c1=" ?*c1*
                                          ", c2=" ?*c2*
                                          ", c3=" ?*c3*   " \" "
                    crlf)

  (printout plotfile "   0.0   0.0 " crlf  ; pool boundary outline
                    " 127.0   0.0 " crlf
                    " 127.0  67.5 " crlf
                    "   0.0  67.5 " crlf
                    "   0.0   0.0 " crlf  " \" \" "
                    crlf)

  (printout auvfile crlf "Environment    nps_pool.off"
                    crlf "Replayfile     " ?filename
                    crlf "Replaysize     " ?*number_of_fields*
                    crlf) ; simulator replay filename/filesize initialization

  (if (or (<> ?*gyroerror* 0.0) (<> ?*gyrodriftrate* 0.0)) then
      (printout plotfile " 100.0 -20.0 " crlf "\"AUV gyro error = "
                         ?*gyroerror*      " degrees, gyro drift rate = "
                         ?*gyrodriftrate*  " degrees/hour \" "
                         crlf))

  (if (or (<> ?*gyroerror* 0.0) (<> ?*gyrodriftrate* 0.0)) then
      (printout auvfile "gyroerror      " ?*gyroerror*     " degrees" crlf
                        "gyrodriftrate  " ?*gyrodriftrate* " degrees/hour"
                         crlf))

  (printout auvfile crlf ?*colorl* "  Color scheme for regression lines "
                    crlf) ; primary default color scheme

  (bind ?*time* (time)) ; start clock timer

  (assert (check-file-flag))
)

;_____________________________________________________________________________

(defrule check-range-file

  ?check-file <- (check-file-flag)
  (not (range-file-closed-flag))
=>
  (retract ?check-file)  ; don't read this file again until point is processed

  (assert (first-element-read-file-flag = (read rangefile)))
  ; first-element-read-file-flag will be asserted with first element from
  ;       the rangefile
)

;_____________________________________________________________________________
```

```
(defrule skip-rangefile-comments  ; keep reading the file until we get a number

  (declare (salience 100))
  ?first-element-read-file <-(first-element-read-file-flag ?file-element & ~EOF)
  (test (not (numberp ?file-element)))
=>
  (retract ?first-element-read-file)
  (printout t ".")
  (readline rangefile)            ; flush comments through end-of-line
  (assert (check-file-flag))
)

;_____________________________________________________________________________

(defrule read-remainder-of-range-record

 (declare (salience 100))
 ?first-element-read-file <- (first-element-read-file-flag ?file-element & ~EOF)
 (test (numberp ?file-element))
=>
  (retract ?first-element-read-file)
  (bind ?field1 ?file-element)
  (bind ?field2  (read rangefile))
  (bind ?field3  (read rangefile))
  (bind ?field4  (read rangefile))
  (bind ?field5  (read rangefile))
  (bind ?field6  (read rangefile))
  (bind ?field7  (read rangefile))
  (bind ?field8  (read rangefile))
  (bind ?field9  (read rangefile))
  (bind ?field10 (read rangefile))
  (bind ?field11 (read rangefile))
  (bind ?field12 (read rangefile))
  (bind ?field13 (read rangefile))
  (bind ?field14 (read rangefile))
  (bind ?field15 (read rangefile))
  (bind ?field16 (read rangefile))
  (bind ?field17 (read rangefile))

  (if (>= ?*number_of_fields* 18) then (bind ?field18 (read rangefile)))
  (if (>= ?*number_of_fields* 19) then (bind ?field19 (read rangefile)))
  (if (>= ?*number_of_fields* 20) then (bind ?field20 (read rangefile)))

 ; account for user-provided gyro error and gyro drift rate:

 (bind ?totalerror (radians (+ ?*gyroerror*
                               (* ?*gyrodriftrate* (/ ?field1 3600.0)))))

  (bind ?heading (- ?field7 ?totalerror))

  ; Don't assert a point if it has no range value (non-return)
  (if (or (> ?field13 1) (> ?field14 1) (> ?field15 1) (> ?field16 1))
   then
       (assert (Range_data (time               ?field1)
                           (x                  ?field2)
                           (y                  ?field3)
                           (z                  ?field4)
                           (phi                ?field5)
                           (theta              ?field6)
                           (psi                ?heading)
                           (p                  ?field8)
                           (q                  ?field9)
                           (r                  ?field10)
                           (delta_dive_planes  ?field11)
                           (delta_rudders      ?field12)
                           (range_a            ?field13)
                           (range_b            ?field14)
                           (range_c            ?field15)
                           (range_d            ?field16)
                           (speed              ?field17))))
  (format ?*out* "%nCompleted reading range record;  data time %3.1f" ?field1)
  (assert (check-file-flag))
)

;_____________________________________________________________________________

(defrule close-range-file

  (declare (salience 100))
  ?first-element-read-file <- (first-element-read-file-flag EOF)
=>
  (retract ?first-element-read-file)
```

```
  (close rangefile)
  (assert (range-file-closed-flag))
  (assert (Point (status NEW)))  ; dummy point so last line (if any) is saved
  (printout t crlf "Closed the input range file." crlf)
)

;_____________________________________________________________________________
;
; Point position calculation functions
;_____________________________________________________________________________
;
; Forward transducer (#1):  reference frame is identical to AUV
; Left    transducer (#2):  psi   = AUV psi   + PI / 2
; Right   transducer (#3):  psi   = AUV psi   - PI / 2
; Depth   transducer (#4):  theta = AUV theta + PI / 2
;_____________________________________________________________________________

(deffunction delta_x (?range ?phi ?theta ?psi)

  (if (= ?*transducer_b*  1)
      then (bind ?result (* ?range (* (cos ?theta) (cos ?psi)))))
  (if (= ?*transducer_b*  2)
      then (bind ?result (* ?range (* (cos ?phi)   (cos (- ?psi (/ (pi) 2)))))))
  (if (= ?*transducer_b*  3)                          ;  | Note caveat about yaw
      then (bind ?result (* ?range (* (cos ?phi)   (cos (+ ?psi (/ (pi) 2)))))))
  (if (= ?*transducer_b*  4)
      then (bind ?result (* ?range (sin ?theta))))
  ?result)

;_____________________________________________________________________________

(deffunction delta_y (?range ?phi ?theta ?psi)
  (if (= ?*transducer_b*  1)
      then (bind ?result (* ?range (* (cos ?theta) (sin ?psi)))))
  (if (= ?*transducer_b*  2)
      then (bind ?result (* ?range (* (cos ?phi)   (sin (- ?psi (/ (pi) 2)))))))
  (if (= ?*transducer_b*  3)                          ;  | Note caveat about yaw
      then (bind ?result (* ?range (* (cos ?phi)   (sin (+ ?psi (/ (pi) 2)))))))
  (if (= ?*transducer_b*  4)
      then (bind ?result (* ?range (sin ?phi))))
  ?result)

;_____________________________________________________________________________

(deffunction delta_z (?range ?phi ?theta ?psi)

  (if (= ?*transducer_b*  1)
      then (bind ?result (* ?range (sin ?theta))))
  (if (= ?*transducer_b*  2)
      then (bind ?result (- 0 (* ?range (sin ?phi)))))
  (if (= ?*transducer_b*  3)
      then (bind ?result (* ?range (sin ?phi))))
  (if (= ?*transducer_b*  4)
      then (bind ?result (* ?range (* (cos ?phi) (cos ?theta)))))
  ?result)

;_____________________________________________________________________________
;
;  Point building rule
;_____________________________________________________________________________

(defrule build-point-from-raw-AUV-range-data

; this rule currently handles only left transducer

  (declare (salience 200))
  ?range_data<- (Range_data (processed FALSE)
                            (time ?time) (x ?x) (y ?y) (z ?z)
                            (phi ?phi) (theta ?theta) (psi ?psi)
                            (range_b ?range)(valid_b ?valid))
  (test (<> ?*transducer_b* 0))
=>
  (bind ?range (* ?range ?*feet_per_sonar_unit*)); unit conversion of range slot
  (bind ?delta_x (delta_x ?range ?phi ?theta ?psi))
  (bind ?delta_y (delta_y ?range ?phi ?theta ?psi))
  (bind ?delta_z (delta_z ?range ?phi ?theta ?psi))
  (if (and (> ?time 0) (> ?range 1)) then ; only make valid data points

     (assert (Point (time ?time)
                    (x =(+ ?x  ?delta_x))
                    (y =(+ ?y  ?delta_y))
                    (z =(+ ?z  ?delta_z))
```

```
                    (valid ?valid)
                    (status NEW)))

;     print sonar return as 'o' and auv position as '*'
      (printout plotfile (+ ?x ?delta_x ?*offsetx*)      " "
                         (+ ?y ?delta_y ?*offsety*) crlf "o" crlf)
      (printout plotfile (+ ?x          ?*offsetx*)      " "
                         (+ ?y          ?*offsety*) crlf "*" crlf)
                       ; include coordinate offsets
      (modify ?range_data (processed TRUE) (range_b ?range))
      (format ?*out* "%nAsserted and plotted a point for data time %3.1f" ?time))
   else  ; a bogus point
      (modify ?range_data (processed TRUE) (range_b ?range))
)

;_____________________________________________________________________________
;
; Two-dimensional parametric regression line analysis rules
;_____________________________________________________________________________

(defrule regression-line-sliding-window-start-criteria

   (declare (salience 300))
   ?start-new-window <- (start-new-window-flag)
   ; Find the next 5 NEW points
   ?point1 <- (Point (status NEW) (time ?time1)(x ?x1)(y ?y1)(z ?z1))
   ?point2 <- (Point (status NEW) (time ?time2)(x ?x2)(y ?y2)(z ?z2))
   ?point3 <- (Point (status NEW) (time ?time3)(x ?x3)(y ?y3)(z ?z3))
   ?point4 <- (Point (status NEW) (time ?time4)(x ?x4)(y ?y4)(z ?z4))
   ?point5 <- (Point (status NEW) (time ?time5)(x ?x5)(y ?y5)(z ?z5))
   (test (< ?time1 ?time2))
   (test (< ?time2 ?time3))
   (test (< ?time3 ?time4))
   (test (< ?time4 ?time5))
=>
   (retract ?start-new-window)
   ; These points are eligible and thus become ACTIVE
   (modify ?point1 (status ACTIVE))
   (modify ?point2 (status ACTIVE))
   (modify ?point3 (status ACTIVE))
   (modify ?point4 (status ACTIVE))
   (modify ?point5 (status ACTIVE))
   (bind ?*n*      5)
   (bind ?*sumx*   (+ ?x1 ?x2 ?x3 ?x4 ?x5))
   (bind ?*sumy*   (+ ?y1 ?y2 ?y3 ?y4 ?y5))
   (bind ?*sumxy*  (+ (* ?x1 ?y1) (* ?x2 ?y2)(* ?x3 ?y3)(* ?x4 ?y4)(* ?x5 ?y5)))
   (bind ?*sumxx*  (+ (* ?x1 ?x1) (* ?x2 ?x2)(* ?x3 ?x3)(* ?x4 ?x4)(* ?x5 ?x5)))
   (bind ?*sumyy*  (+ (* ?y1 ?y1) (* ?y2 ?y2)(* ?y3 ?y3)(* ?y4 ?y4)(* ?y5 ?y5)))
   (bind ?*minz*   (min ?z1 ?z2 ?z3 ?z4 ?z5))
   (bind ?*maxz*   (max ?z1 ?z2 ?z3 ?z4 ?z5))

   (assert (Regression_line (start ?time1) (end ?time5) (status NEW)))
   (format ?*out* "%n%nRegression line sliding window start criteria met.")
)

;_____________________________________________________________________________

(deffunction calculate-line-fit-and-update-global-variables ()

;  global inputs:  n, sumx, sumy, sumxy, sumxx, sumyy

    (bind ?*meanx* (/ ?*sumx* ?*n*))
    (bind ?*meany* (/ ?*sumy* ?*n*))

    (bind ?*sigmaxx* (- ?*sumxx* (/ (* ?*sumx* ?*sumx*) ?*n*)))
    (bind ?*sigmaxy* (- ?*sumxy* (/ (* ?*sumx* ?*sumy*) ?*n*)))
    (bind ?*sigmayy* (- ?*sumyy* (/ (* ?*sumy* ?*sumy*) ?*n*)))

    (bind ?*phi*     (* 0.5 (atan2 (* -2.0 ?*sigmaxy*) (- ?*sigmayy* ?*sigmaxx*))
                     )) ; note paper's caveat re frame of reference of phi

    (bind ?*r*       (+ (* ?*meanx* (cos ?*phi*)) (* ?*meany* (sin ?*phi*))))

    (bind ?term2     (sqrt (+ (* 0.25 (- ?*sigmayy* ?*sigmaxx*)
                                      (- ?*sigmayy* ?*sigmaxx*))
                              (* ?*sigmaxy* ?*sigmaxy*))))

    (bind ?*M-major* (- (/ (+ ?*sigmaxx* ?*sigmayy*) 2.0) ?term2))
    (bind ?*M-minor* (+ (/ (+ ?*sigmaxx* ?*sigmayy*) 2.0) ?term2))

    (bind ?*d-major* (* 4 (sqrt (/ ?*M-minor* ?*n*))))
    (bind ?*d-minor* (* 4 (sqrt (/ ?*M-major* ?*n*))))
```

```
   (bind ?*rho*     (/ ?*d-minor* ?*d-major*))
   (format ?*out* "%nRegression line fit calculations complete.")
)

;_____________________________________________________________________

(defrule regression-line-initial-segment-validity-check

   (declare (salience 300))
   ; Get the NEW Regression_line and 5 ACTIVE Points
   ?line   <- (Regression_line (start ?time1) (end ?time5) (status NEW))

   ?point1 <- (Point (time ?time1) (x ?x1) (y ?y1) (z ?z1) (valid ?valid1))
   ?point5 <- (Point (time ?time5) (x ?x5) (y ?y5) (z ?z5) (valid ?valid5))

   ?point2 <- (Point (time ?time2) (x ?x2) (y ?y2) (z ?z2) (valid ?valid2))
   (test (and (< ?time1 ?time2) (> ?time5 ?time2)))
   ?point3 <- (Point (time ?time3) (x ?x3) (y ?y3) (z ?z3) (valid ?valid3))
   (test (and (< ?time1 ?time3) (> ?time5 ?time3) (<> ?time2 ?time3)))
   ?point4 <- (Point (time ?time4) (x ?x4) (y ?y4) (z ?z4) (valid ?valid4))
   (test (and (< ?time1 ?time4) (> ?time5 ?time4) (<> ?time2 ?time4)
              (<> ?time3 ?time4)))
=>
   (calculate-line-fit-and-update-global-variables)

   (bind ?*rho* (/ ?*d-minor* ?*d-major*))

   (if   (< ?*rho* ?*c3*) ; Validity check:  Test II equation (25)

    then                   ; initial line segment IS valid
         (modify ?point1 (status ENDPOINT))
         (modify ?point2 (status USED))
         (modify ?point3 (status USED))
         (modify ?point4 (status USED))
         (modify ?point5 (status ENDPOINT))
         (modify ?line   (status CURRENT)
                         (r ?*r*)
                         (orientation =(normalize (degrees
                                                   (atan2 (- ?y5 ?y1) (- ?x5 ?x1)))))
                         (correlation ?*rho*))
     (format ?*out* "%nRegression line initial segment validity check passed.")

    else                   ; initial line segment IS NOT valid
         (modify ?point1 (status INVALID))  ; window slides by one to the right
         (modify ?point2 (status NEW))
         (modify ?point3 (status NEW))
         (modify ?point4 (status NEW))
         (modify ?point5 (status NEW))
         (retract ?line)
         (assert (start-new-window-flag))   ; begin building a new window
     (format ?*out* "%nRegression line initial segment validity check failure.")
     (format ?*out* "%n")
   )
)

;_____________________________________________________________________

(defrule regression-line-window-expansion

   (declare (salience 300))
   ; Get the CURRENT Regression_line, start  Point, end Point, and new Point
   ?current-line <- (Regression_line (start ?starttime) (end ?endtime)
                                     (status CURRENT))
   ?new-point    <- (Point (time ?newtime) (x ?newx) (y ?newy)(z ?newz)
                           (status NEW))
   ?start-point  <- (Point (time ?starttime)(x ?startx)(y ?starty)(z ?startz))
   ?end-point    <- (Point (time ?endtime)  (x ?endx)  (y ?endy)  (z ?endz))
=>
  (bind ?*delta* (+ (* (cos ?*phi*)(- ?*meanx* ?newx))
                    (* (sin ?*phi*)(- ?*meany* ?newy))))  ; residual

  (bind ?*sigma* (sqrt (/ ?*M-minor* (- ?*n* 2))))

  (if (and (< ?*delta* (max (* ?*c1* ?*sigma*) ?*c2*)) ; Test I  equation (23)
           (< ?*rho*    ?*c3*)                         ; Test II equation (25)
           (> ?newtime  0))                            ; ignore invalid points


    then                              ;test passed, new point meets criteria
      (modify ?new-point (status USED))  ; we just used this point
      (bind ?*n*     (+ ?*n* 1))
      (bind ?*sumx*  (+ ?*sumx* ?newx))
```

```
  (bind ?*sumy*  (+ ?*sumy* ?newy))
  (bind ?*sumxy* (+ ?*sumxy* (* ?newx ?newy)))
  (bind ?*sumxx* (+ ?*sumxx* (* ?newx ?newx)))
  (bind ?*sumyy* (+ ?*sumyy* (* ?newy ?newy)))
  (bind ?*minz*  (min ?*minz* ?newz))
  (bind ?*maxz*  (max ?*maxz* ?newz))

  ;update globals and then line parameters
  (calculate-line-fit-and-update-global-variables)

  (bind ?correlation    (- 1 ?rho*))

  ; update endpoint status slots for possible retraction of used data
  (modify ?end-point    (status USED))
  (modify ?new-point    (status ENDPOINT))

  (modify ?current-line (r   ?*r*)
          (orientation =(normalize (degrees
                                    (atan2 (- ?newy ?starty) (- ?newx ?startx)))))
                        (correlation ?correlation) ; value range [1-c3..1]
                        (end ?newtime))
  (format ?*out* "       Added another point to the regression line.%n");

else
  (modify ?current-line (status VALID))  ; test failed, save old line

  ;current point retains status NEW unless it is a dummy point at time zero
  (if (= 0 ?newtime) then (retract ?new-point))

  ; initial node of new segment

  (bind ?*delta* (+ (* (cos ?*phi*)(- ?*meanx* ?startx))
                    (* (sin ?*phi*)(- ?*meany* ?starty))))  ; residual

  (bind ?*projection-x*      (+ ?startx (* ?*delta* (cos ?*phi*))))
  (bind ?*projection-y*      (+ ?starty (* ?*delta* (sin ?*phi*))))
  (bind ?start-projection-x  ?*projection-x*)
  (bind ?start-projection-y  ?*projection-y*)
  (bind ?correlation         (- 1 ?*rho*))
  (assert (Node (time ?starttime)           ; edge's virtual start node
                (x          ?*projection-x*)
                (y          ?*projection-y*)
                (z          ?startz)
                (accuracy   ?*d-minor*)     ; minor axis diameter
                (confidence ?correlation))) ; using elliptical thinness
  (format ?*out* "  Valid node completed, data time %3.1f%n" ?starttime)

  (printout plotfile (+ ?*projection-x* ?*offsetx*)  " "
                     (+ ?*projection-y* ?*offsety*)  crlf)
  (format ?*out* "  Projection endpoints (%5.1f, %5.1f)"
                 (+ ?*projection-x* ?*offsetx*)
                 (+ ?*projection-y* ?*offsety*))

  ; final node of new segment

  (bind ?*delta* (+ (* (cos ?*phi*)(- ?*meanx* ?endx))
                    (* (sin ?*phi*)(- ?*meany* ?endy))))  ; residual

  (bind ?*projection-x* (+ ?endx (* ?*delta* (cos ?*phi*))))
  (bind ?*projection-y* (+ ?endy (* ?*delta* (sin ?*phi*))))
  (bind ?confidence     (- 1 ?*rho*))
  (assert (Node (time       ?endtime)       ; edge's virtual end node
                (x          ?*projection-x*)
                (y          ?*projection-y*)
                (z          ?endz)
                (accuracy   ?*d-minor*)     ; minor axis diameter
                (confidence ?confidence)))  ; using elliptical thinness
  (format ?*out* "  (%5.1f, %5.1f)%n"
                 (+ ?*projection-x* ?*offsetx*)
                 (+ ?*projection-y* ?*offsety*))

  (format ?*out* "  Raw data endpoints   (%5.1f, %5.1f)  (%5.1f, %5.1f)%n"
                 (+ ?startx  ?*offsetx*)
                 (+ ?starty  ?*offsety*)
                 (+ ?endx    ?*offsetx*)
                 (+ ?endy    ?*offsety*))
  (format ?*out* "  Valid node completed, data time %3.1f %n" ?endtime)
  (printout plotfile (+ ?*projection-x* ?*offsetx*)  " "
                     (+ ?*projection-y* ?*offsety*)  crlf "\" \"" crlf)

  (assert (Edge (start      ?starttime)
                (end        ?endtime)
```

```
                    (averagez   =(avg ?*minz* ?*maxz*))))
      (format ?*out* "  Valid edge completed, data times (%3.1f .. %3.1f),"
                     ?starttime  ?endtime)
      (format ?*out* " averagez = %3.1f, line r = %3.1f,"
                     (avg ?*minz* ?*maxz*) ?*r*)

      (format ?*out*  " line orientation = %3.1f degrees%n"
              (normalize (degrees (atan2 (- ?newy ?starty) (- ?newx ?startx)))))

      (format auvfile
           "%nPoint   %5.1f %4.1f %3.1f                  time %4.1f"
                      (+ ?start-projection-x ?*offsetx*)
                      (+ ?start-projection-y ?*offsety*)
                      (+ ?startz             ?*offsetz*)
                       ?starttime)
                       ; depth range 0..8 ft, time is optional

      (format auvfile
           "%nPoint   %5.1f %4.1f %3.1f                  time %4.1f"
                      (+ ?*projection-x* ?*offsetx*)
                      (+ ?*projection-y* ?*offsety*)
                      (+ ?endz           ?*offsetz*)
                       ?endtime)

      (format auvfile
           "%nSegment %5.1f %4.1f %3.1f  %5.1f %4.1f %3.1f  time %4.1f"
                      (+ ?start-projection-x   ?*offsetx*)
                      (+ ?start-projection-y   ?*offsety*)
                      (+ (/ (+ ?*minz* ?*maxz*) 2.0) ?*offsetz*)
                      (+ ?*projection-x*       ?*offsetx*)
                      (+ ?*projection-y*       ?*offsety*)
                      (+ (/ (+ ?*minz* ?*maxz*) 2.0) ?*offsetz*)
                       ?endtime)

     (assert (check-file-flag))
     (assert (start-new-window-flag))
     (format ?*out* "  Valid regression line actions completed, data time %3.1f"
                    ?endtime)
     (format ?*out* "%n%n")
    )
)

;_____________________________________________________________________________
;
; Rules for retraction of excess data facts (garbage collection)
;_____________________________________________________________________________

(defrule retract-excess-Range_data

   (retract-excess-data TRUE)
   ?range_data <- (Range_data (processed TRUE))
=>
   (retract ?range_data)
)

(defrule retract-excess-Point

   (retract-excess-data TRUE)
   ?point <- (Point (status INVALID | USED))
=>
   (retract ?point)
)

(defrule retract-excess-endPoint

   (retract-excess-data TRUE)
   ?point <- (Point (status ENDPOINT) (time   ?point-time))
   ?node  <- (Node                    (time   ?node-time))
   (test (= ?point-time ?node-time))
=>
   (retract ?point)
)

;_____________________________________________________________________________
;
; Gyro error rules
;_____________________________________________________________________________

(defrule determine-initial-gyro-error

   (declare (salience 300))
   ?pool <- (location pool) ; this rule only works in the pool
```

```
   ?poly <- (Polyhedron (classification WALL) (start ?polystart) (end ?polyend)
                        (status COMPLETE))
   (test (= ?*newgyroerror* 0.0)) ; first wall provides best est, don't repeat
   ?line <- (Regression_line (start ?start)(end ?end)(orientation ?orientation)
                        (status USED | USED_FOR_AREA))
   (test (= ?polyend ?end))

   ?point1 <- (Point (time ?time1) (x ?x1) (y ?y1) (z ?z1))
   (test (= ?time1 ?start))
   ?point2 <- (Point (time ?time2) (x ?x2) (y ?y2) (z ?z2))
   (test (= ?time2 ?end))
   (test (>= (distance ?x1 ?y1 ?z1 ?x2 ?y2 ?z2) 2.0)) ; skip short segments
=>
   (bind ?deltal (normalize2 (- ?orientation   0.0)))
   (bind ?delta2 (normalize2 (- ?orientation  90.0)))
   (bind ?delta3 (normalize2 (- ?orientation 180.0)))
   (bind ?delta4 (normalize2 (- ?orientation 270.0)))

   (if (< (abs ?deltal) (min (abs ?delta2) (abs ?delta3) (abs ?delta4))) then
       (bind ?*newgyroerror* ?deltal))
   (if (< (abs ?delta2) (min (abs ?deltal) (abs ?delta3) (abs ?delta4))) then
       (bind ?*newgyroerror* ?delta2))
   (if (< (abs ?delta3) (min (abs ?delta2) (abs ?delta1) (abs ?delta4))) then
       (bind ?*newgyroerror* ?delta3))
   (if (< (abs ?delta4) (min (abs ?delta2) (abs ?delta3) (abs ?deltal))) then
       (bind ?*newgyroerror* ?delta4))

   (bind ?*gyroerrortime* (avg ?start ?end)) ; average time of wall segment
   (format t "%nUser-provided gyro error = %4.1f degrees" ?*gyroerror*)
   (format t "%nWall orientation         = %4.1f degrees" ?orientation)
   (format t " for time %3.1f (%3.1f .. %3.1f)"
             (avg ?start ?end)  ?start  ?end)
   (format t "%nExpert system gyro error = %4.1f degrees" ?*newgyroerror*)
)

;_____________________________________________________________________________

(defrule determine-gyro-drift-rate

   (declare (salience 300))
   ?pool <- (location pool) ; this rule only works in the pool
   ?poly <- (Polyhedron (classification WALL) (start ?polystart) (end ?polyend)
                        (status          COMPLETE))
   (test (<> ?*newgyroerror* 0.0)) ; perform only if new gyro error calculated
   ?line <- (Regression_line (start ?start)(end ?end)(orientation ?orientation)
                        (status USED | USED_FOR_AREA))
   (test (= ?polyend ?end))
=>
   (bind ?deltal (normalize2 (- ?orientation ?*newgyroerror*  0.0 )))
   (bind ?delta2 (normalize2 (- ?orientation ?*newgyroerror*  90.0)))
   (bind ?delta3 (normalize2 (- ?orientation ?*newgyroerror* 180.0)))
   (bind ?delta4 (normalize2 (- ?orientation ?*newgyroerror* 270.0)))

   (if (< (abs ?deltal) (min (abs ?delta2) (abs ?delta3) (abs ?delta4))) then
       (bind ?*newgyrodriftrate* ?deltal))
   (if (< (abs ?delta2) (min (abs ?deltal) (abs ?delta3) (abs ?delta4))) then
       (bind ?*newgyrodriftrate* ?delta2))
   (if (< (abs ?delta3) (min (abs ?delta2) (abs ?deltal) (abs ?delta4))) then
       (bind ?*newgyrodriftrate* ?delta3))
   (if (< (abs ?delta4) (min (abs ?delta2) (abs ?delta3) (abs ?delta1))) then
       (bind ?*newgyrodriftrate* ?delta4))

   (format t "%nWall orientation                = %4.1f degrees" ?orientation)
   (format t " for time %3.1f (%3.1f .. %3.1f)"
             (avg ?start ?end)  ?start  ?end)
   (format t "%nCurrent gyro error              = %4.1f degrees%n"
             (+ ?*newgyrodriftrate* ?*newgyroerror*))
   (bind ?*newgyrodriftrate* (* 3600.0 (/ ?*newgyrodriftrate*
                                 (- (avg ?start ?end) ?*gyroerrortime*))))

   (if (<= (abs ?*newgyrodriftrate*) 200.0) then
       (format t "Expert system gyro drift rate = %4.1f degrees/hour %n"
                 ?*newgyrodriftrate*))
)

;_____________________________________________________________________________
;
; Completion!
;_____________________________________________________________________________

(defrule plot-pool-graph-file-when-done  ; this rule is the last to be fired
```

```
   (declare (salience 0))                ; all other rules take precedence
   ?range-file-closed <- (range-file-closed-flag)
=>
   (format t
         "%n%nElapsed time to perform sonar classification:  %3.1f seconds.%n%n"
         (- (time) ?*time*))

   ; all file outputs complete
   (close plotfile)
   (close auvfile)

   (printout t crlf crlf "Sending pool.auv to iris graphics subdirectory."
             crlf)
   ; first save old copy of file to pool.bak
   (printout t crlf "rcp gemini:~brutzman/clips/pool.auv.bak"
                    " iris1:~brutzman/graphics/pool.auv.bak"  crlf)
   (system          "rcp gemini:~brutzman/clips/pool.auv.bak"
                    " iris1:~brutzman/graphics/pool.auv.bak")

   (printout t crlf "rcp gemini:~brutzman/clips/pool.auv"
                    " iris1:~brutzman/graphics/pool.auv"  crlf)
   (system          "rcp gemini:~brutzman/clips/pool.auv"
                    " iris1:~brutzman/graphics/pool.auv")

   (printout t crlf "rcp gemini:~brutzman/clips/pool.auv"
                     " iris1:~brutzman/preview/pool.auv"  crlf)
   (system         "rcp  gemini:~brutzman/clips/pool.auv"
                     " iris1:~brutzman/preview/pool.auv")

    ; You must be running under sunview on a workstation for sunplot to work.
    (printout t crlf crlf "The generated pool.graph sunplot follows:" crlf crlf)

    (printout t "graph -b -g 1 -l \"NPS AUV Sonar Classification Expert System "
              "\" -x 145 -15 -y -50 110  < pool.graph | sunplot -c 650"
              crlf crlf)
    (system     "graph -b -g 1 -l \"NPS AUV Sonar Classification Expert System "
              "\" -x 145 -15 -y -50 110  < pool.graph | sunplot -c 650")
    (system     "rm core") ; remove core dump file which resulted
                           ;   if not running under sunview

  (if   (yes-or-no " Do you want to print the screen log file")
   then (dribble-off)
        (system "enscript -G -r auvsonar.log"))

  (if   (yes-or-no "Do you want a hard copy of the sonar plot")
   then
    (open  "pool.auv"   auvfile  "a")
    (open  "pool.graph" plotfile "a")
    (printout t crlf)
    (if   (yes-or-no "Do you want to add a comment line to the plot")
     then (printout t crlf crlf "Enter comment:  ")
          (bind ?comments (readline))
          (printout   auvfile crlf "Comment:  " ?comments crlf)
          (printout  plotfile " 115 80 " crlf "\"" ?comments " \" " crlf))

    (printout t crlf crlf "The generated pool.graph is being plotted:"
              crlf crlf)
    (printout t "graph -b -g 1 -l \"NPS AUV Sonar Classification Expert System "
              "\" -x 145 -15 -y -50 110  < pool.graph | lpr -g -h -Pap2"
              crlf crlf)
    (system     "graph -b -g 1 -l \"NPS AUV Sonar Classification Expert System "
              "\" -x 145 -15 -y -50 110  < pool.graph | lpr -g -h -Pap2")
    ; all file outputs complete
    (close plotfile)
    (close auvfile)
  )

   (printout t crlf "The generated pool.auv file follows:" crlf crlf)
   (system "more pool.auv")
   (printout t crlf crlf crlf)
)

;-----------------------------------------------------------------------------
;
; Polyhedron output function and rules
;-----------------------------------------------------------------------------

(deffunction output_polyhedron (?starttime ?endtime
                                ?startx    ?starty   ?startz
                                ?endx      ?endy     ?endz
                                ?classification ?comment)
```

```
   (format   t "%n%nThe polyhedron at times (%3.1f .. %3.1f) "
               ?starttime ?endtime)
   (printout t "has classification " ?classification "." crlf crlf)

   (format auvfile
         "%n%s    %5.1f %4.1f %3.1f  %5.1f %4.1f %3.1f  time %4.1f  %s"
                  ?classification
                  (+ ?startx ?*offsetx*)
                  (+ ?starty ?*offsety*)
                  (+ ?startz ?*offsetz*)
                  (+   ?endx ?*offsetx*)
                  (+   ?endy ?*offsety*)
                  (+   ?endz ?*offsetz*)
                  ?endtime
                  ?comment)

     (format auvfile "%n")

     (format ?*out* "%n%n")
     (format ?*out*
          "%n%s    %5.1f %4.1f %3.1f  %5.1f %4.1f %3.1f  time %4.1f  %s"
                  ?classification
                  (+ ?startx ?*offsetx*)
                  (+ ?starty ?*offsety*)
                  (+ ?startz ?*offsetz*)
                  (+   ?endx ?*offsetx*)
                  (+   ?endy ?*offsety*)
                  (+   ?endz ?*offsetz*)
                  ?endtime
                  ?comment)

     (format ?*out* "%n%n")

)

;_____________________________________________________________________________

(defrule change-colors-for-inferred-edges-when-done

   (declare (salience 40))              ; pre-completion rules take precedence
   ?range-file-closed <- (range-file-closed-flag)
=>
   (printout auvfile crlf crlf ?*color2* "  Color scheme for inferred edges "
                   crlf) ; secondary default color scheme
)

;_____________________________________________________________________________

(defrule output-polyhedrons-with-inferred-edges-when-done

   (declare (salience 30))              ; pre-completion rules take precedence
   ?range-file-closed <- (range-file-closed-flag)
   ?poly <- (Polyhedron (status COMPLETE | USED_FOR_AREA)
                        (start   ?startpolytime)
                        (end     ?endpolytime)
                        (startx  ?startx)    (starty ?starty) (startz ?startz)
                        (trait   INFERRED_EDGE)
                        (classification WALL))
;   node matches end of polyhedron
   ?node <- (Node (time  ?nodetime) (x ?nodex) (y ?nodey) (z ?nodez))
   (test (= ?endpolytime ?nodetime))

=>
   (output_polyhedron  ?startpolytime ?endpolytime
                       ?startx ?starty 0.0
                       ?nodex  ?nodey  8.0
                       "WALL"
                       "(inferred edge)")
)

;_____________________________________________________________________________

(defrule change-colors-for-hidden-edges-when-done

   (declare (salience 20))              ; pre-completion rules take precedence
   ?range-file-closed <- (range-file-closed-flag)
=>
   (printout auvfile crlf crlf ?*color3* "  Color scheme for hidden edges "
                   crlf) ; tertiary default color scheme
)

;_____________________________________________________________________________
```

```
(defrule output-object-polyhedrons-with-hidden-edges-when-done

   (declare (salience 10))              ; pre-completion rules take precedence
   ?range-file-closed <- (range-file-closed-flag)
   ?poly <- (Polyhedron (status COMPLETE | USED_FOR_AREA)
                        (start   ?startpolytime)
                        (end     ?endpolytime)
                        (startx  ?startx)    (starty ?starty) (startz ?startz)
                        (trait   HIDDEN_EDGE)
                        (classification WALL))
;   node matches end of polyhedron
   ?node <- (Node (time  ?nodetime) (x ?nodex) (y ?nodey) (z ?nodez))
   (test (= ?endpolytime ?nodetime))

=>
   (output_polyhedron  ?startpolytime ?endpolytime
                       ?startx ?starty 0.0
                       ?nodex  ?nodey  8.0
                       "WALL"
                       "(hidden edge)")
)

;_____________________________________________________________________________
;
; Polyhedron building rules
;_____________________________________________________________________________

(defrule polyhedron-building-start

   (declare (salience 430)) ; lower salience value than polyhedron-building
   ?line <- (Regression_line (status VALID)
                             (start ?starttime) (end ?endtime))

   ?start-node <- (Node      (time     ?nodetime)
                             (accuracy ?accuracy1)
                             (x ?startx) (y ?starty) (z ?startz))
   (test (= ?starttime ?nodetime))

   ?end-node   <- (Node      (time     ?endnodetime)
                             (accuracy ?accuracy2)
                             (x ?endx) (y ?endy) (z ?endz))
   (test (= ?endtime ?endnodetime))
=>
   (assert (Polyhedron (status  ACTIVE)
                       (start  ?starttime) (end    ?endtime)
                       (startx ?startx)    (starty ?starty) (startz ?startz)
                       (centroidx  =(+ ?startx ?endx))
                       (centroidy  =(+ ?starty ?endy))
                       (centroidz  =(+ ?startz ?endz))
                       (sidecount  1)
                       (accuracy   =(max ?accuracy1 ?accuracy2))
                       (trait      OBJECT_BUILDING_BASED)
                       (classification WALL)))

   (modify ?line       (status         USED))

   (bind ?length (distance ?startx ?starty ?startz ?endx ?endy ?endz))

   (if   (>= ?length ?*min_wall_length*)
    then (output_polyhedron  ?starttime ?endtime
                             ?startx ?starty 0.0
                             ?endx   ?endy   8.0
                              WALL "(new polyhedron start)"))

)

;_____________________________________________________________________________

(defrule polyhedron-building-continuation

;  This rule tests the newest edge in relation to the most recent previous edge
;     of the currently ACTIVE polyhedron.
;
;  If the edges are too far apart, the previous polyhedron is COMPLETE and the
;     new edge is ignored in order for it to begin a new polyhedron.
;
;  If the edges are colinear, the new edge is included as part of the
;     currently ACTIVE polyhedron.

;  If the edges are concave, the previous polyhedron is COMPLETE and the new
;     edge is ignored in order to let it begin a new polyhedron.
;
```

```
;  If the edges are convex, the new edge is included as part of the
;     currently ACTIVE polyhedron, and the polyhedron is reclassified
;     from WALL to OBJECT.
;
;  Specific polyhedron OBJECT reclassifications will be made by higher level rules.

   (declare (salience 440)) ; higher salience value than polyhedron start
   ?poly <- (Polyhedron (status      ACTIVE)
                        (start       ?startpolytime)
                        (end         ?endpolytime)
                        (accuracy    ?polyaccuracy)
                        (startx      ?startx)
                        (starty      ?starty)
                        (startz      ?startz)
                        (centroidx   ?centroidx)
                        (centroidy   ?centroidy)
                        (centroidz   ?centroidz)
                        (sidecount   ?sidecount)
                        (area        ?area)
                        (classification     ?classification))

;   linel is most recent valid regression line included in the polyhedron
   ?linel <- (Regression_line (status USED)
                              (start ?startlineltime)
                              (end     ?endlineltime)
                              (orientation ?orientationl))
   (test (= ?endpolytime ?endlineltime))

;   line2 is newest valid regression line to be evaluated
   ?line2 <- (Regression_line (status VALID)
                              (start ?startline2time)
                              (end     ?endline2time)
                              (orientation ?orientation2))

;   nodel matches end of linel (most recent valid regression line)
   ?nodel <- (Node (time  ?nodeltime)
                   (accuracy ?accuracy2)
                   (x ?nodelx) (y ?nodely) (z ?nodelz))
   (test (= ?endlineltime ?nodeltime))

;   node2 matches start of line2
   ?node2 <- (Node (time  ?node2time)
                   (accuracy ?accuracy1)
                   (x ?node2x) (y ?node2y) (z ?node2z))
   (test (= ?startline2time ?node2time))

;   node3 matches end of line2
   ?node3 <- (Node (time  ?node3time)
                   (accuracy ?accuracy3)
                   (x ?node3x) (y ?node3y) (z ?node3z))
   (test (= ?endline2time ?node3time))

=>
   (bind ?distance (distance ?nodelx ?nodely ?nodelz ?node2x ?node2y ?node2z))

;   if distance is too great, don't continue building polyhedron with new edge
   (if   (>  ?distance ?*max_edge_distance*)
    then (modify ?poly (status COMPLETE)
                       (area       =(abs ?area))
                       (centroidx  =(/ ?centroidx ?sidecount 2)) ; 2 points/side
                       (centroidy  =(/ ?centroidy ?sidecount 2))
                       (centroidz  =(/ ?centroidz ?sidecount 2))
                       (sidecounter1   ?sidecount)
                       (sidecounter2   ?sidecount))

        ; if polyhedron was not a WALL, assert a HIDDEN_EDGE wall for it
         (if   (not (eq ?classification WALL))
          then (format t "%nPolyhedron OBJECT (%3.1f .. %3.1f) "
                                    ?startpolytime ?endpolytime)
               (format t "has area %3.1f%n" (abs ?area))
               (format auvfile "  (prior object area was %3.1f)" (abs ?area))
               (assert (Polyhedron (status       COMPLETE)
                                   (start        ?startpolytime)
                                   (end          ?endpolytime)
                                   (startx       ?startx)
                                   (starty       ?starty)
                                   (startz       ?startz)
                                   (centroidx  =(avg ?startx ?node3x))
                                   (centroidy  =(avg ?starty ?node3y))
                                   (centroidz  =(avg ?startz ?node3z))
                                   (sidecount     1)
                                   (accuracy     ?polyaccuracy)
```

```
                                    (trait        HIDDEN_EDGE)
                                    (classification WALL))))
    )

    (bind ?length (distance ?node2x ?node2y ?node2z ?node3x ?node3y ?node3z))

;   if distance is close enough, then test colinear/convex/concave
    (if   (<= ?distance ?*max_edge_distance*)
     then (if   (<= (abs (normalize2 (- ?orientation1 ?orientation2)))
                    ?*max_edge_angle*)
           then ; colinear edge found and added to polyhedron
                ; also add 'S' area between start point and new segments
                (bind ?trianglearea1 (S ?startx ?starty
                                        ?node1x ?nodely ?node2x ?node2y))
                (bind ?trianglearea2 (S ?startx ?starty
                                        ?node2x ?node2y ?node3x ?node3y))
                (modify ?poly  (end   ?endline2time)
                               (area  =(+ ?area ?trianglearea1 ?trianglearea2))
                               (centroidx  =(+ ?centroidx ?node2x ?node3x))
                               (centroidy  =(+ ?centroidy ?node2y ?node3y))
                               (centroidz  =(+ ?centroidz ?node2z ?node3z))

                               (sidecount  =(+ ?sidecount  1)))
                (modify ?line2 (status USED))

                (assert (Polyhedron (status       COMPLETE)
                                    (start       ?endlineltime)
                                    (end         ?startline2time)
                                    (startx      ?node1x)
                                    (starty      ?node1y)
                                    (startz      ?node1z)
                                    (centroidx  =(avg ?node1x ?node2x))
                                    (centroidy  =(avg ?node1y ?node2y))
                                    (centroidz  =(avg ?node1z ?node2z))
                                    (sidecount    1)
                                    (accuracy   =(max ?accuracy1 ?accuracy2))
                                    (trait        INFERRED_EDGE)
                                    (classification WALL)))
                (if   (>= ?length ?*min_wall_length*)
                 then (output_polyhedron  ?startline2time ?endline2time
                                          ?node2x ?node2y 0.0
                                          ?node3x ?node3y 8.0
                                          "WALL"
                                          "(added colinear edge)"))
           else ; test for convex edge to continue building,
                ; otherwise edge is concave and polyhedron is complete.
                ; note this rule currently coded to work only for left transducer

                (if   (< (normalize2 (- ?orientation2 ?orientation1)) 0.0)

                 then ; convex edge found, and joined to polyhedron
                      ; also add 'S' area between start point and new segments
                      (bind ?trianglearea1 (S ?startx ?starty
                                              ?node1x ?nodely ?node2x ?node2y))
                      (bind ?trianglearea2 (S ?startx ?starty
                                              ?node2x ?node2y ?node3x ?node3y))
                      (modify ?poly  (end            ?endline2time)
                                     (classification  OBJECT)
                                     (area       =(+ ?area ?trianglearea1
                                                     ?trianglearea2))
                                     (accuracy   =(max ?accuracy2
                                                       ?accuracy3 ?polyaccuracy))
                                     (centroidx  =(+ ?centroidx ?node2x ?node3x))
                                     (centroidy  =(+ ?centroidy ?node2y ?node3y))
                                     (centroidz  =(+ ?centroidz ?node2z ?node3z))
                                     (sidecount  =(+ ?sidecount  1)))
                      (modify ?line2 (status USED))

                      (assert (Polyhedron (status       COMPLETE)
                                          (start       ?endline1time)
                                          (end         ?startline2time)
                                          (startx      ?node1x)
                                          (starty      ?node1y)
                                          (startz      ?node1z)
                                          (centroidx  =(avg ?node1x ?node2x))
                                          (centroidy  =(avg ?node1y ?node2y))
                                          (centroidz  =(avg ?node1z ?node2z))
                                          (sidecount    1)
                                          (accuracy   =(max ?accuracy1 ?accuracy2))
                                          (trait        INFERRED_EDGE)
                                          (classification WALL)))
```

```
                    (if   (>= ?length ?*min_wall_length*)
                     then (output_polyhedron  ?startline2time ?endline2time
                                              ?node2x ?node2y 0.0
                                              ?node3x ?node3y 8.0
                                              "WALL"
                                              "(added convex edge)"))

               else ; concave edge found so don't continue building polyhedron
                    (modify ?poly (status COMPLETE)
                                  (area       =(abs ?area))
                                  (centroidx  =(/ ?centroidx ?sidecount 2))
                                  (centroidy  =(/ ?centroidy ?sidecount 2))
                                  (centroidz  =(/ ?centroidz ?sidecount 2))
                                  (sidecounter1   ?sidecount)
                                  (sidecounter2   ?sidecount))

                    ; if polyhedron was not a WALL, assert a HIDDEN_EDGE wall
                    (if   (not (eq ?classification WALL))
                     then (format t "%nPolyhedron OBJECT (%3.1f .. %3.1f) "
                                    ?startpolytime ?endpolytime)
                          (format t "has area %3.1f%n" (abs ?area))
                          (format auvfile "  (prior object area was %3.1f)" (abs
?area))
                          (assert (Polyhedron (status  COMPLETE)
                                              (start        ?startpolytime)
                                              (end          ?endpolytime)
                                              (startx       ?startx)
                                              (starty       ?starty)
                                              (startz       ?startz)
                                              (centroidx   =(avg ?startx ?node3x))
                                              (centroidy   =(avg ?starty ?node3y))
                                              (centroidz   =(avg ?startz ?node3z))
                                              (sidecount     1)
                                              (accuracy     ?polyaccuracy)
                                              (trait         HIDDEN_EDGE)
                                              (classification WALL)))))
          )))
)

;_____________________________________________________________________________

(defrule complete-active-polyhedron-after-file-reading-finished

   (declare (salience 420))   ; polyhedron determination rules take precedence
   ?range-file-closed <- (range-file-closed-flag)

   ?poly <- (Polyhedron (status          ACTIVE)
                        (start          ?startpolytime)
                        (end            ?endpolytime)
                        (startx         ?startx)
                        (starty         ?starty)
                        (startz         ?startz)
                        (accuracy       ?polyaccuracy)
                        (centroidx      ?centroidx)
                        (centroidy      ?centroidy)
                        (centroidz      ?centroidz)
                        (sidecount      ?sidecount)
                        (area           ?area)
                        (classification ?classification))

;   node matches end of polyhedron
   ?node <- (Node (time  ?nodetime)
                  (accuracy ?accuracy)
                  (x ?nodex) (y ?nodey) (z ?nodez))
   (test (= ?endpolytime ?nodetime))

=>
    (modify ?poly (status COMPLETE)
                  (area       =(abs ?area))
                  (centroidx  =(/ ?centroidx ?sidecount 2)) ; 2 points/side
                  (centroidy  =(/ ?centroidy ?sidecount 2))
                  (centroidz  =(/ ?centroidz ?sidecount 2))
                  (sidecounter1   ?sidecount)
                  (sidecounter2   ?sidecount))

   ; if polyhedron was not a WALL, assert a HIDDEN_EDGE wall for it
    (if   (not (eq ?classification WALL))
     then (format t "%nPolyhedron OBJECT (%3.1f .. %3.1f) has area %3.1f%n"
                    ?startpolytime ?endpolytime (abs ?area))
          (format auvfile "  (prior object area was %3.1f)" (abs ?area))
          (assert (Polyhedron (status  COMPLETE)
                              (start        ?startpolytime)
```

```
                                (end           ?endpolytime)
                                (startx        ?startx)
                                (starty        ?starty)
                                (startz        ?startz)
                                (centroidx     =(avg ?startx ?nodex))
                                (centroidy     =(avg ?starty ?nodey))
                                (centroidz     =(avg ?startz ?nodez))
                                (sidecount      1)
                                (accuracy      ?polyaccuracy)
                                (trait          HIDDEN_EDGE)
                                (classification WALL)))
)

;_____________________________________________________________________________
;
; Completed polyhedron area calculation rules
;_____________________________________________________________________________

(defrule oldarea1

; compute-polyhedron-area-contribution-from-regression-edges

   (declare (salience 415))   ; polyhedron determination rules take precedence
   ?poly <- (Polyhedron (status         COMPLETE)
                        (trait          OBJECT_BUILDING_BASED)
                        (classification OBJECT)
                        (start         ?startpolytime)
                        (end           ?endpolytime)
                        (startx        ?startx)
                        (starty        ?starty)
                        (startz        ?startz)
                        (centroidx     ?node3x)
                        (centroidy     ?node3y)
                        (centroidz     ?node3z)
                        (sidecount     ?sidecount)
                        (sidecounterl  ?sidecounterl)
                        (area          ?area))

   (test (> ?sidecounterl 0)) ; prevent infinite recursion

;  get the next line contributing to polyhedron area
   ?edge <- (Edge (start      ?startedgetime)
                  (end        ?endedgetime)
                  (status      USED))
   (test (and (>= ?startedgetime ?startpolytime)
              (<= ?endedgetime   ?endpolytime)))

;   nodel matches start of edge
   ?nodel <- (Node (time  ?nodeltime)
                   (x ?nodelx) (y ?nodely) (z ?nodelz))
   (test (= ?startedgetime ?nodeltime))

;   node2 matches end of edge
   ?node2 <- (Node (time  ?node2time)
                   (x ?node2x) (y ?node2y) (z ?node2z))
   (test (= ?endedgetime ?node2time))

=>

   (bind ?trianglearea (S ?nodelx ?nodely ?node2x ?node2y ?node3x ?node3y))

   (modify ?poly (sidecounterl =(- ?sidecounterl 1))
                 (area         =(+ ?area (abs ?trianglearea))))

   (modify ?edge (status        USED_FOR_AREA))

)

;_____________________________________________________________________________

(defrule  oldarea2

; compute-polyhedron-area-contribution-from-inferred-walls

   (declare (salience 410))   ; polyhedron determination rules take precedence
   ?poly <- (Polyhedron (status         COMPLETE)
                        (trait          OBJECT_BUILDING_BASED)
                        (classification OBJECT)
                        (start         ?startpolytime)
                        (end           ?endpolytime)
                        (startx        ?startx)
                        (starty        ?starty)
```

```
                        (startz         ?startz)
                        (centroidx      ?node3x)
                        (centroidy      ?node3y)
                        (centroidz      ?node3z)
                        (sidecount      ?sidecount)
                        (sidecounter2   ?sidecounter2)
                        (area           ?area))

   (test (> ?sidecounter2 0)) ; prevent infinite recursion

;  get a matching inferred wall or hidden edge polyhedron
   ?poly2 <- (Polyhedron (status      COMPLETE)
                         (trait       INFERRED_EDGE | HIDDEN_EDGE)
                         (classification WALL)
                         (start       ?startpoly2time)
                         (end         ?endpoly2time)
                         (startx      ?nodelx)
                         (starty      ?nodely)
                         (startz      ?nodelz))
   (test (and (>= ?startpoly2time ?startpolytime)
              (<= ?endpoly2time   ?endpolytime)))

;   node2 matches the end of this inferred/hidden wall
   ?node2 <- (Node (time  ?node2time)
                   (x ?node2x) (y ?node2y) (z ?node2z))
   (test (= ?endpoly2time ?node2time))

=>
   (bind ?trianglearea (S ?node1x ?nodely ?node2x ?node2y ?node3x ?node3y))

   (modify ?poly   (sidecounter2 =(- ?sidecounter2 1))
                   (area         =(+ ?area (abs ?trianglearea))))

   (modify ?poly2 (status        USED_FOR_AREA))

   (if    (eq ?sidecounter2 1) ; last edge triangle has been added
    then  (format t "%nPolyhedron OBJECT (%3.1f .. %3.1f) has area %3.1f%n"
                  ?startpolytime ?endpolytime (+ ?area (abs ?trianglearea))))

)

;_____________________________________________________________________________
;
; Object classification rules:  the top level at last!
;_____________________________________________________________________________

(defrule classify-pool-objects

   (declare (salience 400))   ; polyhedron determination rules take precedence

   ?poly <- (Polyhedron (status         COMPLETE)
                        (trait          OBJECT_BUILDING_BASED)
                        (classification OBJECT)
                        (start          ?startpolytime)
                        (end            ?endpolytime)
                        (startx         ?startx)
                        (starty         ?starty)
                        (startz         ?startz)
                        (centroidx      ?centroidx)
                        (centroidy      ?centroidy)
                        (centroidz      ?centroidz)
                        (sidecount      ?sidecount)
                        (area           ?area))

;   node matches end of polyhedron
   ?node <- (Node (time     ?nodetime)
                  (accuracy ?accuracy)
                  (x ?endx) (y ?endy) (z ?endz))
   (test (= ?endpolytime ?nodetime))

=>

; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

;  Reclassify long skinny objects as walls

   (bind ?length (distance ?startx ?starty ?startz ?endx ?endy ?endz))
   (if   (<= (/ ?area ?length ?length) ?*wall_thinness_ratio*)
    then (bind  ?area 0.0)
         (modify ?poly (classification WALL) (area 0.0))
         (format t "%n*** OBJECT at (%3.1f .. %3.1f) reclassified as a WALL.%n"
                   ?startpolytime ?endpolytime)
```

```
   )
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

;  Mine classification

   (if   (and (>= ?area 10.0) (<= ?area 100.0)) ; area criteria test
    then (modify ?poly (classification MINE))
         (format   t "%n%nThe polyhedron at times (%3.1f .. %3.1f) "
                     ?startpolytime ?endpolytime)
         (printout t "has classification MINE." crlf crlf)
         (format auvfile
          "%n%s    %5.1f %4.1f %3.1f  %5.1f           time %4.1f"
                   MINE
                   (+ ?centroidx ?*offsetx*)
                   (+ ?centroidy ?*offsety*)
                   (+ ?centroidz ?*offsetz*)
                   (/ ?area (pi) 2 6) ; radius
                   ?endpolytime)
         (format auvfile "%n")

         (format ?*out*
          "%n%s    %5.1f %4.1f %3.1f  %5.1f           time %4.1f"
                   Mine
                   (+ ?centroidx ?*offsetx*)
                   (+ ?centroidy ?*offsety*)
                   (+ ?centroidz ?*offsetz*)
                   (/ ?area (pi) 2 6) ; radius & sonar beamwidth fudge factor
                   ?endpolytime)
         (format ?*out* "%n"))
)

;_____________________________________________________________________________
```

APPENDIX D

Shortest Path Planning in a Circle World

Yutaka Kanayama
Donald P. Brutzman
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
U.S.A.

September 21, 1991

Abstract

This paper discussed the shortest path planning problem in a world with n circular obstacles. This problem can be considered a simplified mathematical model of the shortest path planning problem for a polygonal world. As we established a method to treat circles in a manner similar to nodes in a search graph, the Dijkstra's algorithm can be applied and the problem is solved in time $O(n^2 \log n)$. The A* algorithm dramatically improves searching efficiency in this problem. These results are given with numerous example figures. However, considering the $O(n^2)$ time result on the n line segment visibility problem, there is a strong possibility of being able to solve this circle world problem in $O(n^2)$ time also.

1 Introduction

The shortest path problem in a polygonal world has been extensively discussed in numerous articles. Let n be the number of segments in a polygonal environment. Sharir and Shorr first showed that it is solved in time $O(n^2 \log n)$ [16] and this was improved by Welzl [17] and Asano et.al. [2] to $O(n^2)$. The use of Dijkstra's algorithm to spatial planning problems was first proposed by Lozzano-Perez and Wesley [10].

This paper discusses the shortest path planning problem in a world with n circular obstacles, which first appeared in [12]. This problem can be considered a simplified mathematical model of the

shortest path planning problem for a polygonal world. As is well known, the fundamental approach to solve the shortest path problem is to use visible tangents to obstacles [6, 3, 16, 15, 17, 2, 12, 9, 7]. This paper also uses that approach. One of the significant results of this paper is that we are able to treat a circle in this world like a node in a search graph, and thus are able to employ Dijkstra's algorithm.

At least one previous application, the Stanford cart mobile robot, used a similar circle world model for obstacle representation and avoidance. The combination of vision processing and path planning aboard that small robot proved prohibitive for real-time use due to hardware limitations and algorithmic complexity [12]. It is hoped that the optimal algorithm provided in this paper will support real-time path planning by autonomous vehicles. The circle world search model is directly extendible to the general case of three-dimensional path planning and is particularly suitable for underwater vehicle path planning [5].

2 Problem Statements

The environment for this path planning problem is a plane on which a global Cartesian coordinate system is defined. A circle C is a triple of XY coordinates and a radius (> 0),

$$C = (x, y, r) \tag{1}$$

A world W is a set of circles,

$$W = \{C_1, \cdots, C_n\}, \tag{2}$$

where no pair of circles are intersecting or touching. A *free space* is a complement of the union of all the circles (the free space includes the circular boundaries). Let a start point S and a goal point G be points in the free space. Any path joining S and G must stay in the free space. The problem is to find a shortest path joining S and G in this world W (Fig. 1).

3 Fundamentals

3.1 Tangents and Common Tangents

Let $C = (x, y, r)$ be a circle. If the intersection of a directed line (or a *ray*) and the circle C is only a point, the line is said to be a *tangent* to C. If the circle C is on the left of a tangent to C, the tangent is called a *plus* or *counterclockwise* tangent. If C is on the right of a tangent to C, the tangent is called a *minus* or *clockwise* tangent (Fig. 2). The "plus" or "minus" characteristic of a tangent is called the *mode*.

Let $p_0 \equiv (x_0, y_0)$ be a point outside the circle C. The two tangents from p_0 to C are uniquely determined (Fig 2). The orientation θ^+ of the plus tangent from p_0 to the circle C is

$$\theta^+ = \Psi(p_0, (x, y)) - \sin^{-1}(\frac{r}{dist(p_0, (x, y))}), \tag{3}$$

where $\Psi(p, q)$ stands for the orientation from a point p to point q in four quadrants ($\Psi(p, q) \in [-\pi, \pi]$). On the other hand, the orientation θ^- of the minus tangent from p_0 to the circle C is

$$\theta^- = \Psi(p_0, (x, y)) + \sin^{-1}(\frac{r}{dist(p_0, (x, y))}) \tag{4}$$

Therefore, if a tangent of mode m from a point to a circle C, assuming that its value is +1 if the mode is plus and -1 if it is minus, the previous two equations are expressed by

$$\theta^m = \Psi(p_0, (x, y)) - m \sin^{-1}(\frac{r}{dist(p_0, (x, y))}) = \Psi(p_0, (x, y)) - \sin^{-1}(\frac{mr}{dist(p_0, (x, y))}) \tag{5}$$

The intersection of a tangent and a circle is called an *osculating point.* In this case in Fig 2, the osculating points are also called *landing points.*

Likewise, a tangent of mode m from a circle C to a point p_0 (Fig 3) has an orientation

$$\theta^m = \Psi((x, y), p_0) + \sin^{-1}(\frac{m\, r}{dist(p_0, (x, y))}) \tag{6}$$

In this case, the osculating points are called *leaving points.*

There are four common tangents from a circle $C_1 \equiv (x_1, y_1, r_1)$ to another $C_2 \equiv (x_2, y_2, r_2)$ (Fig. 4).

Each of the four tangents are uniquely named: ++ tangent, +− tangent, −+ tangent and −− tangent. The orientation $\theta^{m1\,m2}$ of these tangents is

$$\theta^{m1\,m2} = \Psi((x_1, y_1), (x_2, y_2)) + \sin^{-1}(\frac{m1\, r_1 - m2\, r_2}{dist(p_0, (x, y))}) \tag{7}$$

Hereafter by "tangent" we mean a directed line segment between a point and osculating point, or between osculating points.

3.2 Visibility

The test for intersection of a (directed or undirected) line segment L with a circle C can be done in a constant time as follows (Fig 6). First we find the region in Fig 6 that the center of C is located in. After that we can compare the radius of C with d, where d is the distance between the center of C and L, or the distance between the center and one of the endpoints of L (if L merely osculates

the circle, it is not considered intersecting). A line segment L in a world W is *visible in a world* W if L does not cross the circumference of any circle in W. Basically, the visibility test in W for a line segment needs $O(n)$ time. However, integrating these tests simply for all the possible tangents requires $O(n^3)$ time, since the number of all the tangents is $O(n^2)$. Thus, we will use some other method as discussed later.

3.3 Characterization of Shortest Path

A *directed circle* is a circle C with a *mode* or *direction*. A mode m is either *counterclockwise* (or *plus*) or *clockwise* (or *minus*). A part of a directed circle can be used as a component of a shortest path.

With a world of W, we define the set $M = M(W)$ of (tangent) modes.

$$M(W) \equiv \{C^+, C^- | C \in W\} \tag{8}$$

A *mode sequence* σ over M is a finite sequence of modes such that no subsequence of C^+C^+, C^+C^-, C^-C^+ or C^-C^- (with a single $C \in W$) is contained. The empty mode sequence is denoted by ϵ. The set of all the mode sequences is expressed as $M(W)^*$ following the convention of language and automata theory. If $W = \{C, D\}$, examples of tangent sequences are:

$$\epsilon, C^+, C^-, D^+, D^-, C^+D^+, C^+D^-, \cdots, D^-C^+, \cdots$$

A mode sequence $\sigma \in M(W)^*$ with a start S and goal G can be interpreted into a path $\pi(\sigma)$ by the following two rules. Furthermore, we assume the same tangent mode does not occur consecutively in a mode sequence σ.

(I) If $\sigma = \epsilon$,

$$\pi(\sigma) = \overline{SG} \text{ if } visible\epsilon(S, G) \tag{9}$$

where $visible\epsilon(S, G)$ means "S and G are visible in this world" for directed line segment $\overline{SG}$. If they are not visible, the value $\pi(\epsilon)$ is undefined.

(II) If $\sigma = C_{i1}^{m1} \cdots C_{iq}^{mq}$, where $q \geq 1$, $C_{i1}, \ldots, C_{iq} \in W$, and $m1, \ldots, mq \in \{+, -\}$,

$$\pi(\sigma) = l_0 k_1 l_1 k_2 l_2 \ldots k_q l_q \tag{10}$$

where

1. the right hand side of this equation is the concatenation of the $2q + 1$ subpaths,
2. l_0 is a $m1$ tangent from S to circle C_{i1} if the tangent is visible (in this world),

3. for each $j(1 \leq j \leq q-1)$, l_j is $(mj, m(j+1))$ common tangent from circle C_{ij} to circle $C_{i,j+1}$ if this tangent is visible,

4. l_q is a mq tangent from circle C_{iq} to G if this tangent is visible, and

5. for each $j(1 \leq j \leq q)$, k_j is the minimum portion of the $m(ij)$-directed circle C_{ij} between the two osculating points of the previous and next tangents (if both tangents are visible).

6. If any of these subpaths do not exist in the world, the value $\pi(\sigma)$ is undefined.

A path π is called *canonical* if there exists a mode sequence σ such that $\pi = \pi(\sigma)$. The following proposition is essential in shortest path planning problem:

Proposition 3.1 *The shortest path for a given world and two endpoints is canonical.*

4 Dijkstra's Algorithm

Having Proposition 3.1, there exists the possibility of applying Dijkstra's algorithm to this path finding problem. Dijkstra's algorithm is a standard approach to the single-source shortest path problem in a graph [1, 11]. Although the free space in a circle world has much more freedom than a network graph in path planning, we can treat a circle as a "generalized" search node and tangents as arcs by using minor search modifications and by taking advantage of Proposition 3.1. This proposition says that a shortest path has a mode sequence σ and lands on one or more circles if S and G are not visible. Therefore, if a circle C^m is in σ, the partial path from S to C^m is the shortest one of all the possible paths from S to C^m. The essence of this concept is the same as that of Dijkstra's algorithm. A relatively complicated question is how two paths with distinct landing points on C should be compared.

The following visibility preprocessing is required prior to employing Dijkstra's algorithm:

1. visible tangents from S to all the circles in W.

2. visible common tangents between all the circles in W.

3. visible tangents from all the circles in W to G.

4.1 Preprocessing for Visible Tangents to or from a Point

Let us consider the problem of finding all the visible tangents from S to all the circles in W. The other problems of finding tangents between circles and tangents to G can be solved in a similar manner.

Step 1. Compute all the orientations of tangents from S to all the circles in both modes.
Step 2. Sort these orientations using a heap.
Step 3. Sweep all the tangent by their orientations starting with the orientation of the tangent with the shortest length. In this sweep process, visibility is tested using a heap built according to tangent lengths. (For sweep technique, see [14].)

The computational time is $O(n)$ for Step 1, $O(n \log n)$ for Step 2, and $O(n \log n)$ for Step 3 (n is the number of circles). Thus, the overall computational complexity is $O(n \log n)$. The result of applying this algorithm to the world given in Figure 1 is shown in Figure 7. All visible tangents from all circles to G are shown in Figure 8.

4.2 Preprocessing for Visible Common Tangents

There are two obvious ways to do this in $O(n^2 \log n)$ time. One is to compute all the common tangents from a circle in $O(n \log n)$ time (Fig 9) and repeat this for all n circles. Another method is to list orientations of all the common tangents among circles. This task also needs $O(n^2 \log n)$ time. Next we sweep these tangents by orientation to update the visibility relation [14, 17]. This part again takes $O(n^2 \log n)$ time.

However, there is a strong possibility of accomplishing the entire process in $O(n^2)$ time using the similar methods developed for finding a visibility graph of n line segments in that time complexity [17, 2]. This is an open problem. All common tangents in the example circle world are shown in Figure 10.

4.3 Comparison between Two Landing Paths

Figure 11 shows an example in which two paths π_1 and π_2 land on a directed circle. In addition let λ_{12} be a portion of the directed circle from the landing point of π_1 to that of π_2. λ_{21} is defined in an analogous manner. The union of both portions make a complete directed circle.

Proposition 4.1 *Assume π_1 and π_2 land on a m-directed circle ($m \in \{+, -\}$). Either of the following cases hold.*

(Case I) A closed path $(\pi_1, \lambda_{12}, \pi_2)$ does not contain the circle.

(Case II) A closed path $(\pi_2, \lambda_{21}, \pi_1)$ does not contain the circle.

Let $lg(\pi)$ denote the length of a path π.

Proposition 4.2 *In the situation described in Proposition 4.1, in Case I, compare*

$$lg(\pi_1) + lg(\lambda_{12}) \text{ and } lg(\pi_2) \tag{11}$$

On the other hand, in Case II, compare

$$lg(\pi_2) + lg(\lambda_{21}) \text{ and } lg(\pi_1) \tag{12}$$

In each case the longer overall path should be discarded.

Proof. Let us examine the situation in Figure 11 which falls under Case I. In this case, we assume the goal G or next landing point is in region L, but not in region R in the figure. If the goal or next landing point is in region R, there exists a better path than π_2 and its use is meaningless. Therefore, the shorter of the two paths in Equation 11 is better for further extension. □

In order to find which of Case I or II in Proposition 4.1 holds, we keep a record of "areas" of paths. Let π be any directed path from a point P to Q. Let us consider a cycle consisting of three paths, the directed line segment $\overline{SP}$ (S is the start of this path planning problem), π itself, and another directed line segment $\overline{QS}$. An *area* $A(\pi)$ of π is an area composed by this cycle. $A(\pi)$ is positive if the cycle is counterclockwise and negative otherwise. Since the algorithm for evaluating $A(\pi)$ is relatively simple, we will not give the details here. Using this area,

Proposition 4.3 *In the situation described in Proposition 4.1,*

- *Assume the mode is +. If*
$$-A(\pi_1) - A(\lambda_{12}) + A(\pi_2) > 0 \tag{13}$$
the closed path falls in Case I. otherwise Case II.
- *Assume the mode is −. If*
$$A(\pi_1) + A(\lambda_{12}) - A(\pi_2) > 0 \tag{14}$$
the closed path falls in Case I. otherwise Case II.

4.4 Dijkstra's Algorithm

Using the preparation so far, applying Dijkstra's algorithm to the shortest path planning problem in a circle world is possible. One of the significant changes from ordinary Dijkstra's algorithm is that the distance from S at a directed circle is related to the landing point to it.

Step 1. Unmark all the directed (means "with mode") circles and let their distance = ∞. Let d_{total} be ∞.
Step 2. Assign the distance of all visible directed circles from S by their distances from S.
Step 3. Select an unmarked directed circle C^m which contains the smallest distance. Mark it.
Step 4. Update the distance to directed circles which are visible from C^m. If a circle already contains a finite distance value, use the comparison algorithm described in Section 4.3.
Step 5. If there is a marked directed circle which is visible to G, update d_{total}.

Step 6. If d_{total} is less than or equal to the distances of all the unmarked directed circles, stop. Otherwise go to Step 3.

The result of applying Dijkstra's algorithm to the sample world is shown in Figure 12.

5 A* Algorithm

In this problem, the use of the A* algorithm is natural and effective [13]. Consider a partial path π whose last landing point is P on a directed circle C^m. A lower estimate of the length for the rest of the path π (or a heuristic for π) is the direct distance between P and the goal G. However, there is a better heuristic function.

First, take a tangent from C^m to G. Let Q be the leaving point on C^m. The heuristic function is the sum of a partial directed circle PQ and the Euclidean distance $\overline{QG}$ (Fig. 13). In finding the tangent from C^m to G, their is no need to calculate visibility. Even if the tangent from C^m to G is not visible, this is a "lower" estimate and a valid heuristic. The result of applying this heuristic function is depicted in Figure 14.

References

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structures and Algorithms*, Reading, Massachusetts, Addison-Wesley, 1985.

[2] Asano, T., Asano T., Guibas, L., Hershberger, J. and Imai, H., "Visibility-Polygon Search and Euclidean Shortest Paths," *Proceedings 26th Symposium on Foundations of Computer Science* pp. 155-164, 1985.

[3] Chazelle, B. and Guibas, L.J., "Visibility and Intersection Problems in Plane Geometry," *Proceedings of the ACM Symposium on Computational Geometry*, 1985.

[4] Chew, L.P., "Planning the Shortest Path for a Disc in $O(n^2 logn)$ Time," *Proceedings of the ACM Symposium on Computational Geometry*, pp. 214-220, 1985.

[5] Healey, A.J., McGhee, R.B., Christi, R., Papoulias, F.A., Kwak, S.H., Kanayama and Y., Lee, Y., "Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle," *Proc. Workshop on Mobile Robots for Subsea Environments, IARP*, in Monterey, California, pp. 177-186, October 23-26, 1990

[6] Hershberger, J. and Guibas, L.J., "An $O(n^2)$ Shortest Path Algorithm for a Non-rotating Convex Body," *Journal of Algorithms*, vol. 9, pp. 18-46, 1988.

[7] Kanayama, Y., "Introduction to Spatial Reasoning," Class Notes, Naval Postgraduate School, Monterey, California, May 1991.

[8] Khatib, O., "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *The International Journal of Robotics Research*, vol.5, No.1, pp90-98, 1986.

[9] Liu, Y.H. and Arimoto, S., "Proposal of Tangent Graph and Extended Tangent Graph for Path Planning of Mobile Robots," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 312-317, 1991.

[10] Lozzano-Perez, T. and Wesley, M.A., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *CACM* vol. 22, no. 10, Oct. 1979, pp. 560-570.

[11] Manber, Udi, *Introduction to Algorithms: A Creative Approach*, New York, Addison-Wesley, pp. 204-208, 1989.

[12] Moravec, Hans P., "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover," *Ph.D. Thesis, Report STAN-CS-80-813*, Stanford University, September 1980.

[13] Nilsson, N.J., *Artificial Intelligence*, Palo Alto, California, Tioga Press, 1980.

[14] Preparata, F.P., and Shamos, M.I., *Computational Geometry: An Introduction*, pp. 10-11, Springer-Verlag, New York, 1985.

[15] Reif, J. and Storer, J.A., "Shortest Paths in Euclidean Space with Polyhedral Obstacles," *Technical Report S-85-121* , Brandeis University, 1985.

[16] Sharir, M. and Schorr, A., "On Shortest Paths in Polyhedral Spaces," *Technical Report No. 138* , New York University.Courant Institute of Mathematical Sciences, 1984.

[17] Welzl, E., "Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time," *Information Processing Letters*, vol. 20, no. 4, pp. 167-171, 1985.

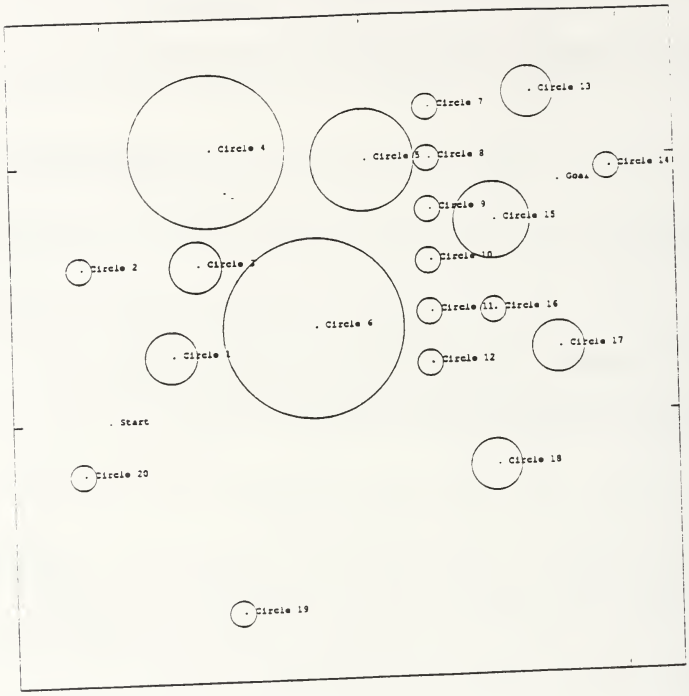


Figure 1: Circle World

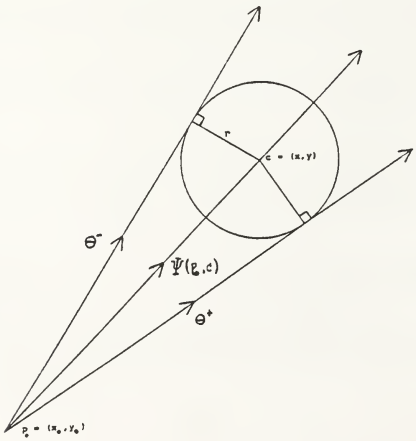


Figure 2: Tangents from Point to Circle

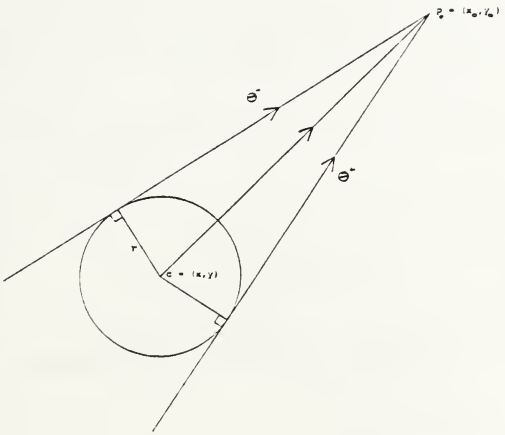


Figure 3: Tangents from Circle to Point

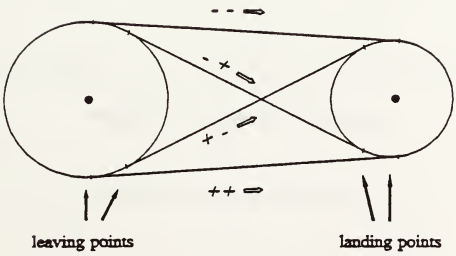


Figure 4: Tangents from Circle to Circle

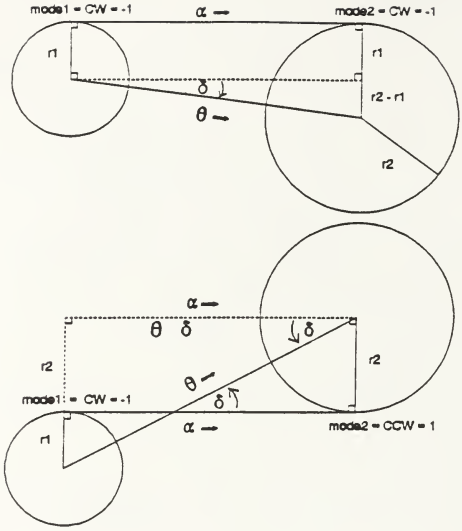
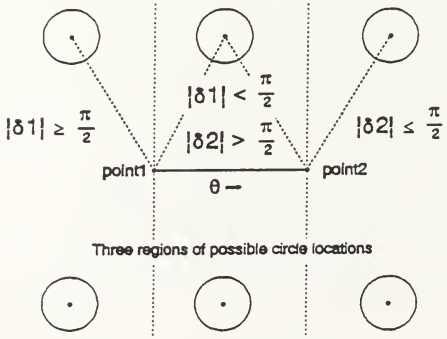


Figure 5: Cross and External Tangents

Point-to-Point Visiblity Checks



θ = orientation (point1, point2)
δ1 = orientation (point1, circle.center) - θ
δ2 = orientation (point2, circle.center) - θ

Figure 6: Intersection Test

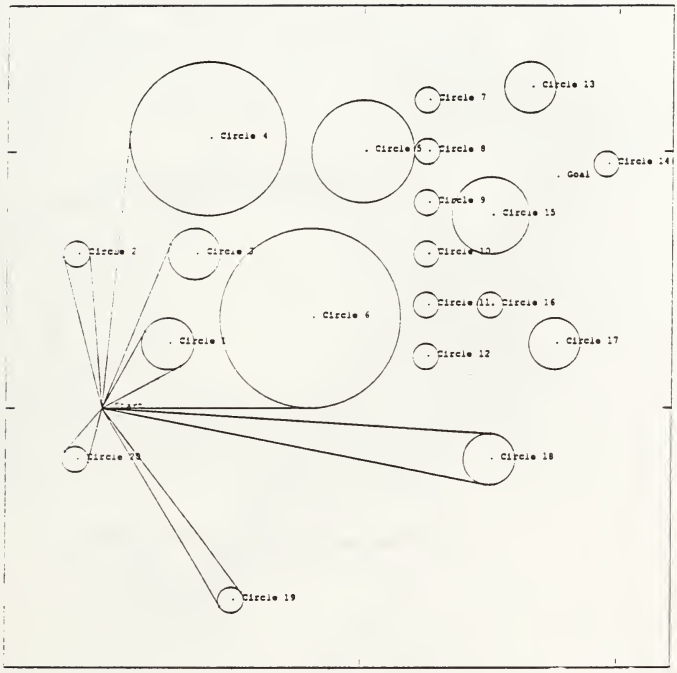


Figure 7: Tangents from Start

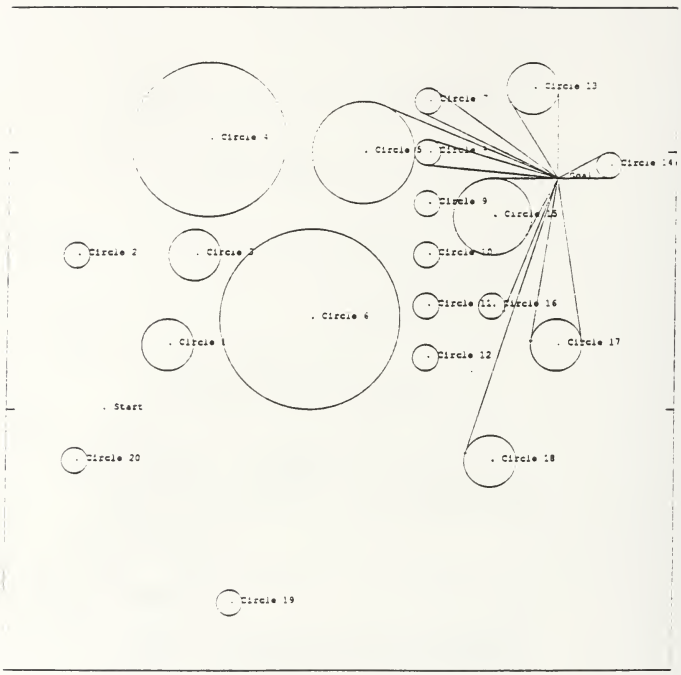


Figure 8: Tangents to Goal

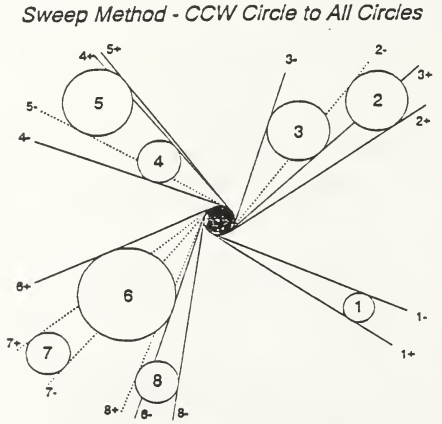


Figure 9: Common Tangents from One Circle

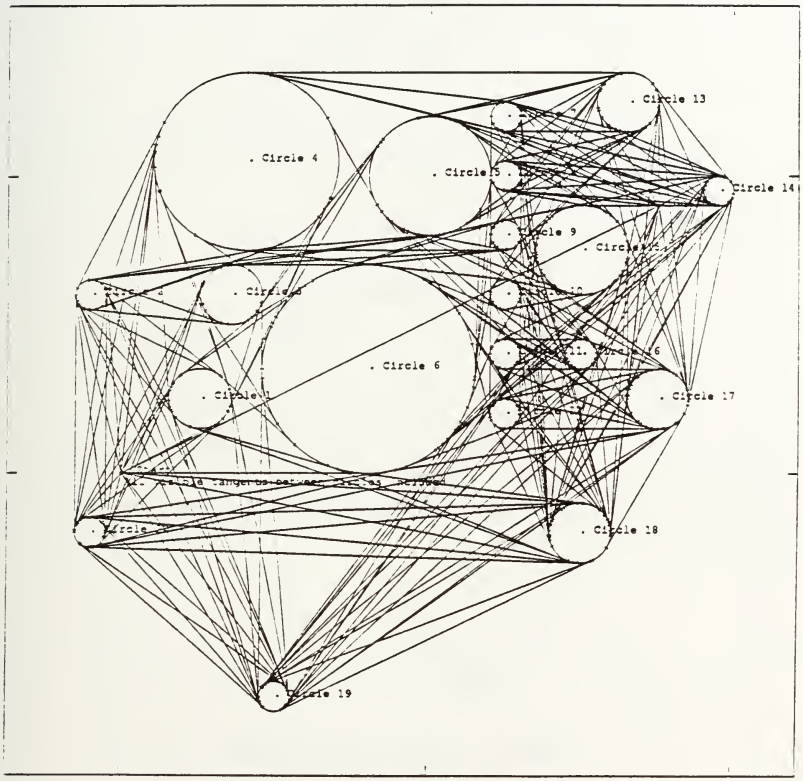


Figure 10: All Common Tangents

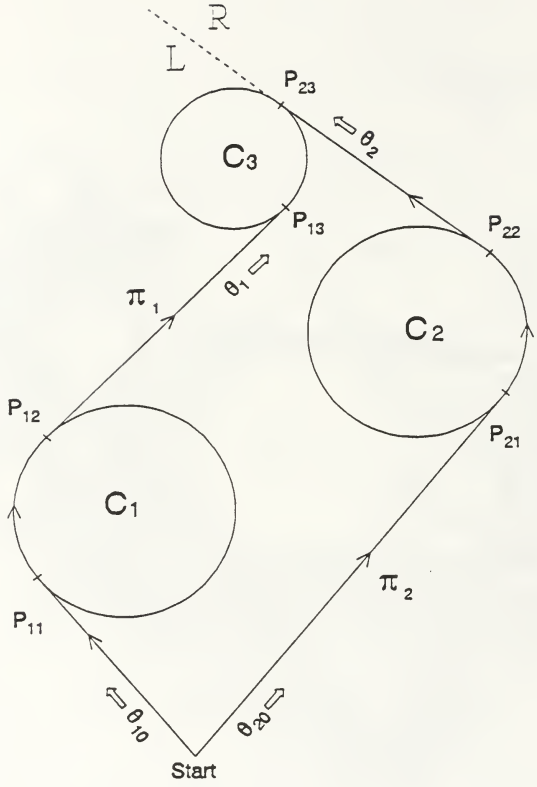


Figure 11: Multiple Paths to Circle

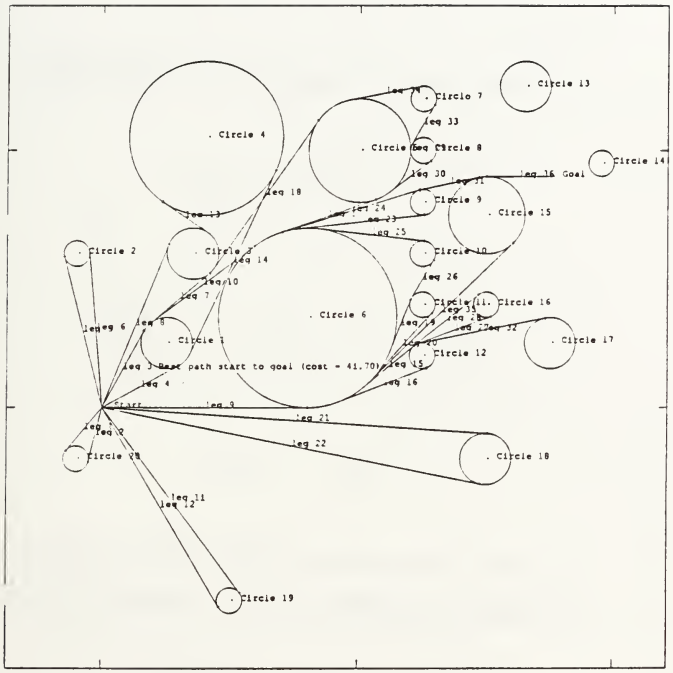
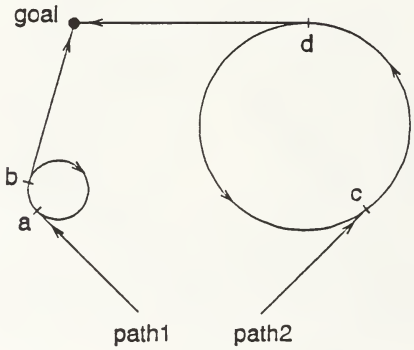


Figure 12: Result: Dijkstra's Algorithm

A^* Evaluation Function Comparison



cost (path1) + arc_cost (a, b, CW) + distance (b, goal) $\overset{?}{>}$
cost (path2) + arc_cost (c, d, CCW) + distance (d, goal)

Figure 13: Heuristic Function

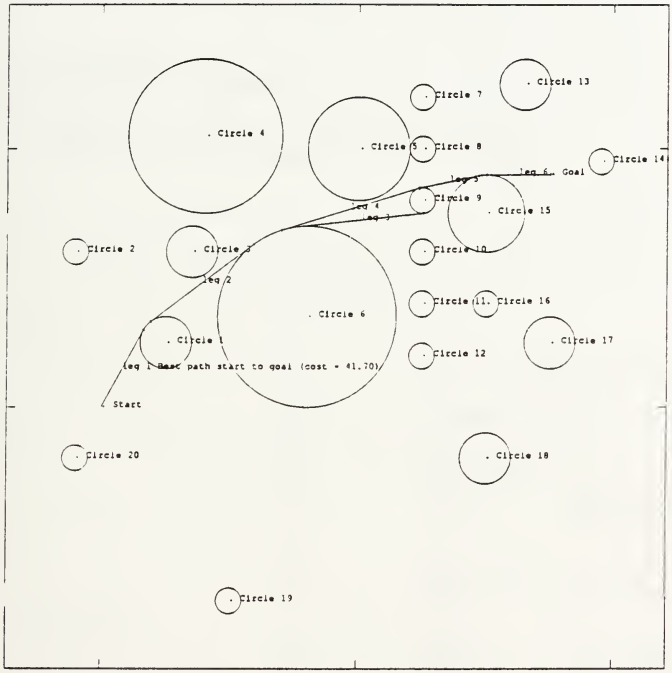


Figure 14: Result: A* Algorithm

APPENDIX E. CIRCLE WORLD SOURCE CODE

```
/*****************************************************************************
*                                                                            *
*   Filename:   circle.c                                                     *
*                                                                            *
*   Purpose:    Define structures, type definitions and functions for        *
*               circle_world robotics project.                               *
*                                                                            *
*   Reference:  Advanced Robotics class notes, Dr. Yutaka Kanayama           *
*                                                                            *
*   Author:     Don Brutzman                                                 *
*                                                                            *
*   Date:       10 February 92                                               *
*                                                                            *
*   Comments:   circle_world is a set of routines for mobile robot modeling  *
*                  and two-dimensional path planning.                        *
*                                                                            *
*               All obstacles are modeled as circles.                        *
*               Circles are allowed to be adjacent but not overlapping.      *
*               Adjacent (touching) circles do not prevent robot travel      *
*               along either circle circumference.                           *
*                                                                            *
*   Language:   ANSI C                                                       *
*                                                                            *
*   Compile:    cc -g -c circle.c -lm                                        *
*                                                                            *
*   Graphing:   graph  -b  -g 1  -l "circle_world plot"  <  filename  | lpr -g *
*                                                                            *
*   Status:     Shortest path solution complete.                             *
*                                                                            *
*****************************************************************************/


#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define CIRCLE.C_INCLUDED


/****** Circle_world Global Constants ************************************/

#define CCW              1
#define PLUS             1
#define RIGHT            1
#define POSITIVE         1

#define CW              -1
#define MINUS           -1
#define LEFT            -1
#define NEGATIVE        -1

#define COLINEAR         0
#define CENTER           0
#define ZERO             0

#define TRUE             1
#define SUCCESS          1
#define YES              1
#define ON               1

#define FALSE            0
#define FAILURE          0
#define NO               0
#define OFF              0

#define VISIBLE          1
#define NONVISIBLE       0
#define TANGENTIAL      -1

#define FATAL            1
#define NONFATAL         0
#define UNDEFINED       -1

#define PI               3.141592653589793
```

```
#define TANGENTS_OK       TRUE    /* whether or not tangents which pass adjacent*/
                                  /*   to other circles are considered VISIBLE  */

#define EPSILON           1.0E-6 /* error bound in floating point calculations */

#define ARC_FACTOR        1.00    /* factor that arc lays outside circle plot   */

#define TICK_WIDTH        0.50    /* tick width at segment endpoints along path */

#define GRAPH_STRETCH     1.20    /* expansion factor to open out graph window  */

#define SUBDIVISIONS      360     /* number of points used to graphically       */
                                  /*   represent a circle during file output    */

#define DEFAULT_Z         0.0     /* Default pool depth, legal range 0..8 feet  */
                                  /*   where zero feet = surface                */
                                  /* makes circle.auv output data compatible    */
                                  /*   with NPS_Pool_Preview graphics project.  */

#define GRAPH_FILENAME    "circle.graph"   /* name of graph points output file */

#define AUV_FILENAME      "circle.auv"     /* name of high level output file   */

#define TRACE             FALSE  /* Enable trace printf statements in circle.c */

static float pooltime  = 0.0;    /* used to put time hacks on output objects   */


/*** List of circle_world Data Structures ************************************/
/*

     External Data Structures     Data Types and Member Labels
     _________________________    ___________________________

     Point                        double          x, y;

     Segment                      Point           point1, point2;

     Circle                       Point           center;
                                  double          radius;

     Tangent                      Circle          circle;
                                  double          angle;

     Arc                          Circle          circle;
                                  double          angle1, angle2;
                                  int             rotation;

    *Configuration                Tangent         tangent;
                                  double          orientation;

    *Path                         char           *label;
                                  Segment         initial_segment;
                                  int             degree;
                                  Path_list      *path_list;

    *Circle_world                 Point           start, goal;
                                  int             degree;
                                  Circle_list    *circle_list;


     Internal Data Structures
     ________________________

    *Path_list                    Arc             arc;
                                  Segment         segment;
                                  Path_list      *previous, *next;

    *Circle_list                  Circle          circle;
                                  Circle_list    *previous, *next;

*/


/*** List of circle_world  Functions *****************************************/
/*

     Functions             Parameters
     _________             __________
```

```
    error                   (message, fatal)
    make_point              (x, y)
    make_segment            (point1, point2)
    make_circle             (point, radius);
    make_tangent            (circle, angle)
    make_arc                (circle, angle1, angle2, rotation)
    create_configuration    (tangent, angle, configuration)
   *create_path             (initial_segment)
    create_circle_world     (start, goal, circle_world)

    sign                    (x)
    degrees                 (angle)
    normalize               (angle)
    normalize2              (angle)
    precede                 (angle1, angle2)
    orientation             (point1, point2)
    distance                (point1, point2)

    angle                   (point1, point2, point3)
    S                       (point1, point2, point3)
    area                    (point1, point2, point3)
    order                   (point1, point2, point3)
    between                 (point1, point2, point3)

    circumference_point     (circle, angle)
    intersect               (segment, circle)
    visible                 (point1, point2, circle_world)

    circle_tangent          (circle1, circle2, model, mode2, *config1, *config2)
    arc_cost                (arc)
    segment_cost            (segment)

    augment_path            (arc, segment, path)
    add_circle_to_world     (circle, circle_world)
    find_circle             (n, circle_world)
    retrieve_circle_world   (circle_world)

    graph_path              (path, circle_world, filename)
    graph_world             (circle_world, filename)
    output_path             (path, filename)
    output_world            (circle_world, filename)
    center_graph_window     (filename, xminptr, xmaxptr, yminptr, ymaxptr,
                             magnification)

  See c_search.c (circle_search) for additional data structures and functions.

*/


/****** Circle_world Data Structures and Type Definitions ********************/

/*---------------------------------------------------------------------------*/

typedef struct Point_type
{
  double x, y;                  /* cartesian coordinate system on a 2D plane  */
}
  Point;

/*---------------------------------------------------------------------------*/

typedef struct Segment_type
{
  Point point1, point2;         /* Note that a Segment is NOT just a          */
}                               /*   collection of (double) x, y coordinate   */
  Segment;                      /*   pairs, i.e. a segment is a pair of Points*/

/*---------------------------------------------------------------------------*/

typedef struct Circle_type
{
  Point  center;
  double radius;                /* radius zero circles are considered points  */
}
  Circle;

/*---------------------------------------------------------------------------*/

typedef struct Tangent_type     /* all circle_world angles are in RADIANS     */
{
```

```
  Circle circle;
  double angle;                 /* angle points to the circle tangent point   */
                                /*     relative to the circle center          */
}
  Tangent;

/*--------------------------------------------------------------------------*/

typedef struct Arc_type
{
  Circle circle;
  double angle1,                /* angle1 points to landing point on circle   */
         angle2;                /* angle2 points to leaving point on circle   */
  int    rotation;              /* direction of rotation is either CW or CCW: */
}                               /* RIGHT(+) landing tangent => CCW(+) rotation*/
  Arc;                          /* LEFT (-) landing tangent => CW (-) rotation*/

/*--------------------------------------------------------------------------*/

typedef struct Configuration_type
{
  Tangent tangent;
  double  orientation;          /* orientation is the tangential orientation, */
}                               /*  i.e. the direction that robot is pointing */
  Configuration;

/*--------------------------------------------------------------------------*/

typedef struct Path_list_type
{
  Arc       arc;                /* arc is from landing point to leaving point */
  Segment   segment;            /* segment connects leaving point, either to  */
                                /*   next circle landing point, or to goal    */
  struct    Path_list_type      /* continue this type with doubly-linked list */
           *previous,           /*   of arc/segment combinations until the    */
           *next;               /*   goal is reached                          */
}
  Path_list;

/*--------------------------------------------------------------------------*/

typedef struct Path_type
{
  char        *label;           /* optional string label for each path        */
  Segment      initial_segment; /* initial segment leaving start point        */
  int          degree;          /* number of arc/segment pairs in the path    */
  Path_list   *path_list;       /* linked list of arc/segment combinations    */
}
  Path;

/*--------------------------------------------------------------------------*/

typedef struct Circle_list_type
{
  Circle      circle;
  struct      Circle_list_type  /* continue this type with  doubly-linked     */
             *previous,         /*   list of remaining circles                */
             *next;
}
  Circle_list;

/*--------------------------------------------------------------------------*/

typedef struct Circle_world_type
{
  Point        start;           /* starting point                             */
  Point        goal;            /* goal     point                             */
  int          degree;          /* number of circles in this world            */
  Circle_list *circle_list;     /* linked list of circles in world            */
}
  Circle_world;
/*--------------------------------------------------------------------------*/


/****** circle_world Function Declarations ***********************************/

/*--------------------------------------------------------------------------*/

void   error (message, fatal)        /* internal error-handling diagnostic    */

       char *message;                /* error message to be printed           */
```

```
      int   fatal;                  /* exit if FATAL, return otherwise       */
{
  fprintf (stderr, "*** Program error: ");
  perror  (message);
  fprintf (stderr, "\n\n");
  if (fatal == FATAL)
  {
     fprintf (stderr, "FATAL error, program exit.\n");
     exit (FAILURE);
  }
  return;
}
/*-------------------------------------------------------------------------------*/

Point make_point (x, y)           /* make a point from x, y coordinates       */

       double x, y;
{
    Point point;

    point.x = x;
    point.y = y;

    return point;
}
/*-------------------------------------------------------------------------------*/

Segment  make_segment (point1, point2)      /* make a Segment from two points */

         Point point1, point2;
{
    Segment segment;

    segment.point1 = point1;
    segment.point2 = point2;

    return segment;
}
/*-------------------------------------------------------------------------------*/

Circle  make_circle (point, radius) /* create a circle from point & radius */

         Point  point;
         double radius;
{
    Circle circle;

    circle.center = point;
    circle.radius = radius;

    return circle;
}
/*-------------------------------------------------------------------------------*/

Tangent  make_tangent (circle, angle)    /* make tangent from circle & angle  */

         Circle circle;
         double angle;
{
    Tangent tangent;

    tangent.circle = circle;
    tangent.angle  = angle;

    return tangent;
}
/*-------------------------------------------------------------------------------*/

Arc   make_arc (circle, angle1, angle2, rotation)

      Circle  circle;               /* create an arc_type from circle data,  */
      double  angle1, angle2;       /* two angles and a direction of rotation*/
      int     rotation;             /* (CW or CCW)                           */
{
    Arc arc;

    /* initialize values */

    arc.circle   = circle;
    arc.angle1   = angle1;
    arc.angle2   = angle2;
    arc.rotation = rotation;
```

```
    return arc;
}
/*--------------------------------------------------------------------------*/

void create_configuration (tangent, orientation, configuration)

     Tangent         tangent;          /* create a configuration_type pointer */
     double          orientation;      /*   from a tangent and an orientation */
     Configuration  *configuration;    /* resulting configuration output      */

{
    if (configuration == NULL)         /* allocate memory if needed           */
    {
        if (TRACE) printf ("\n*** create_configuration:  allocating memory");
        if ((configuration = (Configuration *) malloc (sizeof (Configuration)))
                           == NULL)
            error ("create_configuration:  memory allocation failure", FATAL);
    }

    /* initialize values */

    configuration->tangent     = tangent;
    configuration->orientation = orientation;

    return;
}
/*--------------------------------------------------------------------------*/

static Path  *create_path (initial_segment)

       Segment initial_segment;      /* begin a path_type linked list, using  */
                                     /* a segment which includes start point  */
{
    static Path *path;

    if ((path = (Path *) malloc (sizeof (Path))) == NULL)
        error ("create_path:  memory allocation failure", FATAL);

    /* initialize values */

    path->label             = " ";
    path->degree            = 0;
    path->initial_segment   = initial_segment;
    path->path_list         = ((Path_list *) 0);

    return path;
}
/*--------------------------------------------------------------------------*/

void create_circle_world (start, goal, world)

     Point          start, goal;      /* create a circle_world using          */
                                      /*   start and goal points only         */
     Circle_world  *world;            /* resulting circle_world output        */
{
    if (world == NULL)                /* allocate memory if needed            */
    {
        if (TRACE) printf ("\n*** create_circle_world:  allocating memory");
        if ((world = (Circle_world *) malloc (sizeof (Circle_world))) == NULL)
            error ("create_circle_world:  memory allocation failure", FATAL);
    }

    /* initialize values */

    world->start       = start;
    world->goal        = goal;
    world->degree      = 0;
    world->circle_list = ((Circle_list *) 0);

    return;
}
/*--------------------------------------------------------------------------*/

int    sign (x)                       /* return sign of x as an integer       */
       double x;
{
    if       (x > 0.0)
      return POSITIVE;
    else if  (x < 0.0)
      return NEGATIVE;
    else
      return ZERO;
```

```
}
/*--------------------------------------------------------------------------*/

double degrees (angle)                  /* conversion from radians to degrees    */
       double angle;                    /* note no normalization is performed    */
{
    return angle * 180.0 / PI;
}
/*--------------------------------------------------------------------------*/

double normalize (angle)                /* standard normalization range -PI..PI  */

       double angle;                    /* angle is any real value in RADIANS    */
{
    double x;
    x = angle;
    while (x >   PI) x = x - PI - PI;
    while (x < - PI) x = x + PI + PI;
    return x;
}
/*--------------------------------------------------------------------------*/

double normalize2 (angle)               /* alternate normalization range 0..2*PI */

       double angle;                    /* angle is any real value in RADIANS    */
{
    double x;
    x = angle;
    while (x > PI + PI) x = x - PI - PI;
    while (x < 0.0)     x = x + PI + PI;
    return x;
}
/*--------------------------------------------------------------------------*/

int    precede (angle1, angle2)         /* Boolean function for angle precedence */

       double angle1, angle2;           /* angles are any real angles in degrees */
                                        /* return TRUE if angle1 precedes angle2,*/
                                        /*   return FALSE otherwise              */
{
/* note that the input angles are individually normalized to ensure validity  */

    if (normalize (normalize (angle2) - normalize (angle1)) > 0.0)
       return TRUE;
    else                                /* reference:  equation (7) class notes  */
       return FALSE;
}
/*--------------------------------------------------------------------------*/

double orientation (point1, point2)     /* range -PI .. PI                       */

       Point point1, point2;            /* return normalized angle between points*/
{
    if   ((point1.x == point2.x) && (point1.y == point2.y))
          return 0.0;
    else
          return normalize (atan2 (point2.y - point1.y, point2.x - point1.x));
}
/*--------------------------------------------------------------------------*/

double distance (point1, point2)        /* euclidean distance between two points */
       Point point1, point2;
{
    double deltax = point2.x - point1.x;
    double deltay = point2.y - point1.y;
    return sqrt (deltax * deltax + deltay * deltay);
}
/*--------------------------------------------------------------------------*/

int angle (point1, point2, point3)      /* return angle between three points     */
                                        /* angle order is point1..point2..point3 */
    Point  point1, point2, point3;

{
    return normalize (orientation (point2, point1) -
                      orientation (point2, point3));
}
/*--------------------------------------------------------------------------*/

double S (point1, point2, point3)       /* calculate S function for three points */
                                        /* reference:  equation (14) class notes */
    Point  point1, point2, point3;
```

```
/* CCW triples are positive & CW triples are negative, matching conventions.  */

{
    return 0.5 * ((point2.x - point1.x) * (point3.y - point1.y) -
                  (point3.x - point1.x) * (point2.y - point1.y));
}
/*--------------------------------------------------------------------------*/

double area (point1, point2, point3) /* calculate area between three points   */
                                     /* reference: Proposition 3.1 class notes*/
    Point  point1, point2, point3;
{
    return fabs (S (point1, point2, point3));
}
/*--------------------------------------------------------------------------*/

int    order (point1, point2, point3)/* determine order of three points       */

       Point  point1, point2, point3;/* relationship between three points is  */
                                     /* clockwise (CW), counterclockwise (CCW)*/
                                     /*   or COLINEAR                         */
                                     /* reference:  equation (15) class notes */
{
    if (fabs (S (point1, point2, point3)) <= EPSILON)
       return COLINEAR;              /* floating point error check, correction*/
    else
       return sign (S (point1, point2, point3));
}
/*--------------------------------------------------------------------------*/

int between (point1, point2, point3) /* test for betweenness of point2        */

    Point  point1, point2, point3;
{
    if ((order (point1, point2, point3) == COLINEAR)     &&
        (point1.x <= point2.x) && (point2.x <= point3.x) &&
        (point1.y <= point2.y) && (point2.y <= point3.y))
        return TRUE;
    else
        return FALSE;
}
/*--------------------------------------------------------------------------*/

Point  circumference_point (circle, angle) /* return coordinates of a point   */
                                           /*   on a circle's circumference   */

       Circle circle;  double angle;
{
    return make_point (circle.center.x + circle.radius * cos (angle),
                       circle.center.y + circle.radius * sin (angle));
}
/*--------------------------------------------------------------------------*/

int intersect (segment, circle)     /* determine if segment intersects circle */

    Segment      segment;
    Circle       circle;

    /************************************************************************/
    /* return: TRUE       if any  intersection exists within the circle     */
    /* return: FALSE      if no   intersections exist within the circle     */
    /* return: TANGENTIAL if only intersection is within EPSILON of the     */
    /*                            circle circumference (i.e. tangential),   */
    /*                            TANGENTS_OK is defined as TRUE, & neither */
    /*                            segment endpoint is on the circumference. */
    /************************************************************************/
{
    /* local variable declarations                                          */
    double       height,             /* height of circle center from segment  */
                 theta,              /* orientation angle of segment          */
                 difference;         /* difference between circle.radius and  */
                                     /* distance of circle.center to segment  */

    /* check special case:  circle center is on the line segment            */
    if (between (segment.point1, circle.center, segment.point2))
       {
        if (TRACE) {printf ("\n*** intersect:  betweenness case  ");
                    printf ("(%3.1f, %3.1f) (%3.1f, %3.1f)",
                    segment.point1.x, segment.point1.y,
                    segment.point2.x, segment.point2.y);}
        if  (circle.radius <= EPSILON)   /* this accounts for "point" circles */
            return TANGENTIAL;
```

```
        else
            return TRUE;                    /* intersection occurred            */
        }

    /* determine theta = orientation angle of segment                              */
    theta = orientation (segment.point1, segment.point2);

    if (TRACE) printf ("\n*** intersect:  ");

          /* check case where circle.center is on right hand side of segment  */
    if    (fabs (normalize (orientation (segment.point2, circle.center) - theta))
              <= (PI/2.0))
          {
          difference = distance (segment.point2, circle.center) - circle.radius;
          if (TRACE) printf ("case 1  ");
          if (TRACE) printf ("orientation = %f  ",
                 degrees (orientation (segment.point2, circle.center)));
          if (TRACE) printf (" theta = %f  ", theta);
          if (TRACE) printf (" normalize = %f  ",
             degrees (normalize (orientation (segment.point2, circle.center))));
          }

    else  /* check case where circle.center is on left hand side of segment   */
       if (fabs (normalize (orientation (segment.point1, circle.center) - theta))
                >= (PI/2.0))
          {
          difference = distance (segment.point1, circle.center) - circle.radius;
          if (TRACE) printf ("case 2  ");
          }

    else  /* check case where segment length is zero                           */
       if (distance (segment.point1, segment.point2) == 0.0)
          {
          difference = distance (segment.point1, circle.center) - circle.radius;
          if (TRACE) printf ("case 3  ");
          }

    else  /* circle.center is in a voronoi region defined by segment edge      */
          {
          height     = area (segment.point1, segment.point2, circle.center)
                       * 2 / distance (segment.point1, segment.point2);
          difference = height - circle.radius;
          if (TRACE) printf ("case 4  ");
          }

    /*       Now use 'difference' to test for tangency or intersection.       */

    if (TRACE) {printf ("difference = %f  ", difference);
                printf ("(%3.1f, %3.1f) (%3.1f, %3.1f)",
                         segment.point1.x, segment.point1.y,
                         segment.point2.x, segment.point2.y);
                printf ("  circle (%3.1f, %3.1f) ",
                         circle.center.x, circle.center.y);
               }

    if      (difference < - EPSILON) /* circle.radius is greater than the     */
            return TRUE;             /*   circle's distance from the segment  */
                                     /*   thus intersection is TRUE           */

    else if ((fabs (distance (segment.point1, circle.center) - circle.radius)
                <= EPSILON) ||
             (fabs (distance (segment.point2, circle.center) - circle.radius)
                <= EPSILON))
            return FALSE; /* ignore external tangency of segment endpoints,   */
                          /*   because solely trying to determine tangency    */
                          /*   with different circles in circle_world         */

    else if ((fabs (difference) <= EPSILON) && TANGENTS_OK == TRUE)
            return TANGENTIAL;

    else
            return FALSE;            /* circle does not intersect segment     */
}
/*---------------------------------------------------------------------------*/

int visible (point1, point2, circle_world)

    Point          point1, point2;   /* determine whether a direct path is    */
    Circle_world  *circle_world;     /*   visible between two points without  */
                                     /*   crossing any circles in circle_world*/

    /*************************************************************************/
```

```
    /* return:    VISIBLE if no   intersections exist with circle_world       */
    /* return: NONVISIBLE if any  intersection exists with circle_world       */
    /* return: TANGENTIAL if only intersection(s) are within EPSILON of       */
    /*                           circle circumference(s), i.e. tangential     */
    /*************************************************************************/
{
    /* local variable declarations                                           */
    int          i,                 /* index                                 */
                 visibility;        /* VISIBLE, NONVISIBLE or TANGENTIAL     */
    Segment      segment;           /* intersection of perpendicular & line  */
    Circle       circle;            /* local variable holding current circle */
    Circle_list *world_ptr;         /* pointer to current circle             */

    visibility = VISIBLE;           /* default initialization                */
    segment    = make_segment (point1, point2);
    world_ptr  = circle_world->circle_list; /* first circle in world         */

    /* Note that intersection with circle_world start & goal is not checked.  */

    for (i=1; i <= circle_world->degree; ++i)
    {
        /* check next circle in circle_world for intersections               */
        circle = world_ptr->circle;

        if       (intersect (segment, circle) == TANGENTIAL)
                 visibility = TANGENTIAL; /* i.e. close enough to be a tangent*/
                                          /*    continue searching           */
        else if  (intersect (segment, circle) == TRUE)
             {
                if (TRACE) {printf ("\n*** visible complete:  NONVISIBLE\n");}
                visibility = NONVISIBLE;
                return visibility;
             }
        world_ptr = world_ptr->next;
    }
    if       ((visibility == VISIBLE) && TRACE)
              printf ("\n*** visible complete:  VISIBLE\n");

    else if ((visibility == TANGENTIAL) && TRACE)
              printf ("\n*** visible complete:  TANGENTIAL\n");

    return visibility;
}
/*-------------------------------------------------------------------------*/

void circle_tangent (circle1, circle2, model, mode2, config1, config2)

     Circle          circle1,  /*  input: Leaving circle where tangent starts */
                     circle2;  /*  input: Landing circle where tangent ends   */
     int             model,    /*  input: which side of circle1               */
                     mode2;    /*  input: which side of circle2               */
     Configuration  *config1,  /* output: starting configuration pointer      */
                    *config2;  /* output:   ending configuration pointer      */
{
    double  alpha, theta, delta, angle1, angle2; /* local declarations        */
    Circle  circle3;           /*  input: Leaving circle where tangent starts */
    Circle  circle4;           /*  input: Leaving circle where tangent starts */

    circle3 = circle1;
    circle4 = circle2;

    theta = orientation (circle1.center, circle2.center);

    /* Simplified delta angle equation originated by LT Scott Starsman USN    */

    delta = asin ((mode2 * circle2.radius - model * circle1.radius) /
                  distance (circle2.center, circle1.center));

    alpha = normalize (theta - delta);          /* tangential orientation     */

    angle1 = normalize (alpha - model * PI / 2); /* leaving angle circle1     */
    angle2 = normalize (alpha - mode2 * PI / 2); /* landing angle circle2     */

    if (model == CENTER)
    {
        angle1         = 0.0;
        circle3.radius = 0.0;
    }
    if (mode2 == CENTER)
    {
        angle2         = 0.0;
        circle4.radius = 0.0;
```

```
    }

    if (TRACE)
       printf ("\n*** circle_tangents:  theta = %f, delta = %f, alpha = %f, \n",
               degrees (theta), degrees (delta), degrees (alpha));
    if (TRACE) printf ("                          angle1 = %f, angle2 = %f\n",
               degrees (angle1), degrees (angle2));

    config1->tangent     = make_tangent (circle3, angle1);
    config1->orientation = alpha;
    config2->tangent     = make_tangent (circle4, angle2);
    config2->orientation = alpha;

    return;
}
/*-------------------------------------------------------------------------------*/

double arc_cost (arc)                  /* euclidean distance cost function      */

       Arc arc;                        /* rotation direction is included in arc */
{
    double delta_angle;

    if      (arc.rotation == CW)
       delta_angle = normalize (arc.angle1) - normalize (arc.angle2);

    else if (arc.rotation == CCW)
       delta_angle = normalize (arc.angle2) - normalize (arc.angle1);

    else if (arc.rotation == ZERO)
       delta_angle = 0.0;

    else
    {
       delta_angle = normalize (arc.angle1) - normalize (arc.angle2);
       printf ("\nIllegal rotation value (%ld) given to arc_cost function.",
               "  Assumed CLOCKWISE.\n", arc.rotation);
    }

    /* circumference portion = 2 * PI * R * (delta_angle / (2 * PI))          */
    return arc.circle.radius * normalize2 (delta_angle);
}
/*-------------------------------------------------------------------------------*/

double segment_cost (segment)          /* euclidean distance cost function      */

       Segment segment;
{
    return distance (segment.point1, segment.point2);
}
/*-------------------------------------------------------------------------------*/

void   augment_path (arc, segment, path)

       Arc             arc;           /* add arc and segment to path           */
       Segment         segment;       /* the order of adding an arc followed   */
       Path           *path;          /* by a segment is a rigorous requirement*/
{
    Path_list         *path_ptr;      /* index pointer to legs on the path     */
    Path_list         *path_node;     /* local variable to build path leg      */

    if ((path_node = (Path_list *) malloc (sizeof (Path_list))) == NULL)
        error ("augment_path:  memory allocation failure!", FATAL);

    /* initialize values of path_node which will augment current path         */
    path_node->arc      = arc;
    path_node->segment  = segment;
    path_node->next     = ((Path_list *) 0);
    path_node->previous = ((Path_list *) 0);

    if (path->degree == 0)             /* first path in path_list to be added   */
        path->path_list = path_node;

    else
    {
        /* point to first leg of path, then find end of current path_list     */
        path_ptr = path->path_list;
        while (path_ptr->next != ((Path_list *) 0))
               path_ptr = path_ptr->next;

        /* now augment current path with new path leg                         */
        path_node->previous = path_ptr;
```

```
        path_ptr ->next      = path_node;
    }
    path->degree++;
    if (TRACE)
       printf("\n*** path->degree = %i, augment_path complete\n", path->degree);
    return;
}
/*--------------------------------------------------------------------------*/

void   add_circle_to_world (circle, circle_world)

       Circle         circle;        /* circle to be added to world          */
       Circle_world  *circle_world;  /* current circle_world                 */
{
    Circle_list      *circle_ptr;    /* index pointer to current circle      */
    Circle_list      *circle_node;   /* local variable to build circle leg   */
    double            separation;    /* used to check & prevent circle overlap*/

    if (TRACE) printf ("\n*** add_circle_to_world start\n");

    if ((circle_node = (Circle_list *) malloc (sizeof (Circle_list))) == NULL)
        error ("add_circle_to_world:  memory allocation failure!", FATAL);

    separation = distance (circle.center, circle_world->start);
    if ((separation - circle.radius) < - EPSILON)
          {
            printf("\n*** add_circle_to_world:  the new circle at (");
            printf("%4.2f, %4.2f) \n", circle.center.x,
                                       circle.center.y);
            printf("is not being added because it overlaps the start point");
            printf(" at (%4.2f, %4.2f) \n\n", circle_world->start.x,
                                              circle_world->start.y);
            return;
          }

    separation = distance (circle.center, circle_world->goal);
    if ((separation - circle.radius) < - EPSILON)
          {
            printf("\n*** add_circle_to_world:  the new circle at (");
            printf("%4.2f, %4.2f) \n", circle.center.x,
                                       circle.center.y);
            printf("is not being added because it overlaps the goal point");
            printf(" at (%4.2f, %4.2f) \n\n", circle_world->goal.x,
                                              circle_world->goal.y);
            return;
          }

    /* initialize values of circle_node which will be added to circle_world   */
    circle_node->circle           = circle;
    circle_node->next             = ((Circle_list *) 0);
    circle_node->previous         = ((Circle_list *) 0);

    if (circle_world->degree == 0) /* first circle in circle_world to be added*/
        circle_world->circle_list = circle_node;

    else
    {
        /* point to first circle in world, then find end of current circles   */
        circle_ptr = circle_world->circle_list;

        while (circle_ptr->next != ((Circle_list *) 0))
        {
          separation = distance (circle.center, circle_ptr->circle.center);
          if ((separation - circle.radius - circle_ptr->circle.radius)
               < - EPSILON)
          {
            printf("\n*** add_circle_to_world:  the new circle at (");
            printf("%4.2f, %4.2f) \n", circle.center.x,
                                       circle.center.y);
            printf("is not being added because it overlaps the circle");
            printf(" at (%4.2f, %4.2f) \n\n", circle_ptr->circle.center.x,
                                              circle_ptr->circle.center.y);
            return;
          }
          circle_ptr = circle_ptr->next;
        }
        /* now check that pesky last circle                                   */
        separation = distance (circle.center, circle_ptr->circle.center);
        if ((separation - circle.radius - circle_ptr->circle.radius)
             < - EPSILON)
        {
            printf("\n*** add_circle_to_world:  the new circle at (");
```

```
            printf("%4.2f, %4.2f) \n", circle.center.x,
                                       circle.center.y);
            printf("is not being added because it overlaps the circle");
            printf(" at (%4.2f, %4.2f) \n\n", circle_ptr->circle.center.x,
                                              circle_ptr->circle.center.y);
           return;
        }
        /* now add new circle_node to current circle_list in circle world     */
        circle_node->previous = circle_ptr;
        circle_ptr ->next     = circle_node;
    }
    circle_world->degree++;
    if (TRACE) printf ("\n*** circle_world->degree = %i\n",
                       circle_world->degree);
    if (TRACE) printf ("\n*** add_circle_to_world complete\n");

    return;
}
/*-------------------------------------------------------------------------------*/

Circle  find_circle (n, circle_world)

        int             n;          /* get the nth circle from circle_world */
        Circle_world   *circle_world;
{
        int          i, nn;
        Circle       circle0;
        Circle_list *circle_ptr;
        nn = n;

        if (circle_world->degree == 0)
        {
            printf ("\n*** find_circle:  there are no circles in circle_world;",
                    "\n    using the start point as a zero-radius circle.\n");
            circle0 = make_circle (circle_world->start, 0.0);
            return circle0;
        }
        while ((nn <= 0) || (nn > circle_world->degree))
        {
            printf ("\n*** find_circle: there are %d circles in circle_world.",
                     circle_world->degree);
            printf ("   Which do you want?  ");
            scanf ("%d", &nn);  printf ("\n");
        }
        /* ready to go; point to first circle in world, then find n_th circle */
        i = 1;
        circle_ptr = circle_world->circle_list;
        while ((i < nn) && (circle_ptr->next != NULL))
        {
              circle_ptr = circle_ptr->next;
              ++i;
        }
        return circle_ptr->circle;
}
/*-------------------------------------------------------------------------------*/

void   graph_path (path, circle_world, filename)

       /* Print path data for unix 'graph' use, appended to 'filename'        */

       Path          *path;
       Circle_world  *circle_world;
       char          *filename;
{
    FILE        *file_ptr;
    double       begin_angle, end_angle, delta_angle, angle,
                 midpoint_x, midpoint_y, x1, y1, x2, y2;
    int          i, j, n, rotation;  /* indices, # arc steps & local variable */
    Point        point;
    Circle       arc_circle;
    Path_list   *path_ptr;           /* index pointer to legs on the path     */

    if (TRACE) printf ("\n*** graph_path start\n");

    if (path == NULL) printf ("\n*** path == NULL, error!\n");

     path_ptr = path->path_list;       /* point to first leg of path           */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("graph_path:  file open failure!", NONFATAL);
        return;
```

```
}
/* print starting line segment                                              */
if (TRACE) printf (" %f %f\n",          path->initial_segment.point1.x,
                                        path->initial_segment.point1.y);
if (TRACE) printf (" %f %f\n\" \"\n", path->initial_segment.point2.x,
                                        path->initial_segment.point2.y);
fprintf (file_ptr, " %f %f\n",          path->initial_segment.point1.x,
                                        path->initial_segment.point1.y);
fprintf (file_ptr, " %f %f\n\" \"\n", path->initial_segment.point2.x,
                                        path->initial_segment.point2.y);

/* Print tick marks perpendicular to endpoint of initial segment  */
if (((path->initial_segment.point2.x != circle_world->start.x)  ||
     (path->initial_segment.point2.y != circle_world->start.y)) &&
    ((path->initial_segment.point2.x != circle_world->goal.x)   ||
     (path->initial_segment.point2.y != circle_world->goal.y)))
{
   angle = orientation (path->initial_segment.point1,
                        path->initial_segment.point2) + (PI/2.0);
   x1 = path->initial_segment.point2.x - (TICK_WIDTH / 2.0) * cos (angle);
   y1 = path->initial_segment.point2.y - (TICK_WIDTH / 2.0) * sin (angle);
   x2 = path->initial_segment.point2.x + (TICK_WIDTH / 2.0) * cos (angle);
   y2 = path->initial_segment.point2.y + (TICK_WIDTH / 2.0) * sin (angle);
   fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
   if (TRACE) printf ("tickmark at end of initial segment: \n");
   if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
}
/* Print path label adjacent to midpoint of initial_segment                 */
midpoint_x = (path->initial_segment.point1.x +
              path->initial_segment.point2.x) / 2.0;
midpoint_y = (path->initial_segment.point1.y +
              path->initial_segment.point2.y) / 2.0;
if (TRACE) printf (" %f %f\n", midpoint_x, midpoint_y);
fprintf (file_ptr, " %f %f\n", midpoint_x, midpoint_y);

if (TRACE)
{
    if (path->label != NULL)  printf ("\"%s \"\n", path->label);
    else                      printf ("\" \"\n");
}
if (path->label != NULL)      fprintf (file_ptr, "\"%s \"\n", path->label);
else                          fprintf (file_ptr, "\" \"\n");

/* print all succeeding arc / line segment combinations                     */
for (i=1; i <= path->degree; ++i, path_ptr = path_ptr->next)
{
    /* Calculate and plot arc traversal points                              */


    begin_angle = normalize2 (path_ptr->arc.angle1);
    end_angle   = normalize2 (path_ptr->arc.angle2);
    rotation    =             path_ptr->arc.rotation;

    if      (fabs (begin_angle - end_angle) <= EPSILON)
             delta_angle = 0.0;
    else if ((precede (end_angle, begin_angle) && (rotation == CCW)) ||
             (precede (begin_angle, end_angle) && (rotation == CW)))
             delta_angle = PI + PI - fabs (end_angle - begin_angle);
    else
             delta_angle =     rotation * (end_angle - begin_angle);

    delta_angle = normalize2 (delta_angle);
    angle       = begin_angle;
    arc_circle  = path_ptr->arc.circle;

    n = (int)((float)(SUBDIVISIONS) * fabs (delta_angle) / (PI + PI) + .5);
    if   ((delta_angle == 0.0) || (n <= 0) || (rotation == CENTER))
          n = 0;  /* perform only one iteration of loop                     */
    else
    {
       /* Print first point of arc without ARC_FACTOR for continuity        */
       point = circumference_point (arc_circle, begin_angle),
       fprintf (file_ptr, " %f %f\n", point.x, point.y);
       if (TRACE) printf (" %f %f\n", point.x, point.y);
    }

    if (TRACE)
    {
        printf ("\n*** graph_path  n = %d, delta_angle = %f\n",
                 n, degrees(delta_angle));
        printf (  "                 begin_angle = %f, ",
                    degrees (begin_angle));
```

```
            printf ("end_angle = %f, rotation = %d\n\n",
                        degrees (end_angle), rotation);
        }

        /* Factor radius to graph arc just outside circle circumference       */
        arc_circle.radius *= ARC_FACTOR;

        /* calculate and print points for the arc starting from initial angle */
        for (j = 0; j <= n; ++j)
        {
            if (TRACE) printf ("*** graph_path:  j = %d, angle = %f, n = %d \n",
                            j, degrees (angle), n);
            point = circumference_point (arc_circle, angle);

            if  (n != 0)  fprintf (file_ptr, " %f %f\n", point.x, point.y);
            if ((n != 0) && (TRACE)) printf (" %f %f\n", point.x, point.y);
            if  (n != 0) angle += (rotation * delta_angle / (double) n);
            angle = normalize2 (angle);
        }
        /* calculate and print points for the segment following the arc       */
        point = path_ptr->segment.point1;
        fprintf (file_ptr, " %f %f\n", point.x, point.y);
        if (TRACE) printf (" %f %f\n", point.x, point.y);

        point = path_ptr->segment.point2;
        fprintf (file_ptr, " %f %f\n\" \"\n", point.x, point.y);
        if (TRACE) printf (" %f %f\n\" \"\n", point.x, point.y);

        /* Print tick mark perpendicular to start point of current segment  */
        if (((path_ptr->segment.point1.x != circle_world->start.x)  ||
             (path_ptr->segment.point1.y != circle_world->start.y)) &&
            ((path_ptr->segment.point1.x != circle_world->goal.x)   ||
             (path_ptr->segment.point1.y != circle_world->goal.y)))
        {
           angle = orientation (path_ptr->segment.point1,
                                path_ptr->segment.point2) + (PI/2.0);
           xl = path_ptr->segment.point1.x - (TICK_WIDTH / 2.0) * cos (angle);
           yl = path_ptr->segment.point1.y - (TICK_WIDTH / 2.0) * sin (angle);
           x2 = path_ptr->segment.point1.x + (TICK_WIDTH / 2.0) * cos (angle);
           y2 = path_ptr->segment.point1.y + (TICK_WIDTH / 2.0) * sin (angle);
           fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", xl, yl, x2, y2);
           if (TRACE) printf ("tickmark at start of current segment: \n");
           if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", xl, yl, x2, y2);
        }
        /* Print tick mark perpendicular to final point of current segment  */
        if (((path_ptr->segment.point2.x != circle_world->start.x)  ||
             (path_ptr->segment.point2.y != circle_world->start.y)) &&
            ((path_ptr->segment.point2.x != circle_world->goal.x)   ||
             (path_ptr->segment.point2.y != circle_world->goal.y)))
        {
           angle = orientation (path_ptr->segment.point1,
                                path_ptr->segment.point2) + (PI/2.0);
           xl = path_ptr->segment.point2.x - (TICK_WIDTH / 2.0) * cos (angle);
           yl = path_ptr->segment.point2.y - (TICK_WIDTH / 2.0) * sin (angle);
           x2 = path_ptr->segment.point2.x + (TICK_WIDTH / 2.0) * cos (angle);
           y2 = path_ptr->segment.point2.y + (TICK_WIDTH / 2.0) * sin (angle);
           fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", xl, yl, x2, y2);
           if (TRACE) printf ("tickmark at end of current segment: \n");
           if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", xl, yl, x2, y2);
        }
    }
    fclose (file_ptr);
    if (TRACE) printf ("\n*** graph_path complete\n");
    return;
}
/*-------------------------------------------------------------------------*/

void    graph_world (circle_world, filename)

        /* Print circle_world data for unix 'graph' use, appended to filename */

        Circle_world  *circle_world;
        char          *filename;
{
    FILE        *file_ptr;
    int          i, j;                      /* indices                     */

    Circle_list *circle_ptr;         /* index pointer to current circle    */
    if (TRACE) printf ("\n*** graph_world start\n");
    circle_ptr = circle_world->circle_list; /* point to first circle in world */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
```

```
    {
        error ("graph_world:  file open failure!", NONFATAL);
        return;
    }

    if (TRACE) printf(" %f %f\n", circle_world->start.x, circle_world->start.y);
    if (TRACE) printf("\". Start\"\n");
    fprintf(file_ptr, " %f %f\n", circle_world->start.x, circle_world->start.y);
    fprintf(file_ptr, "\". Start\"\n");

    if (TRACE) printf (" %f %f\n", circle_world->goal.x, circle_world->goal.y);
    if (TRACE) printf ("\". Goal\"\n");
    fprintf(file_ptr, " %f %f\n", circle_world->goal.x, circle_world->goal.y);
    fprintf(file_ptr, "\". Goal\"\n");

    /* Loop to graph all circles in circle_world.                              */
    for (i=1; i <= circle_world->degree; ++i, circle_ptr = circle_ptr->next)
    {
        /* print current circle center                                         */
        if (TRACE) printf (" %f %f\n", circle_ptr->circle.center.x,
                                       circle_ptr->circle.center.y);
        fprintf (file_ptr, " %f %f\n", circle_ptr->circle.center.x,
                                       circle_ptr->circle.center.y);
        if (TRACE) printf ("\". Circle %d\"\n", i); /* label center w/ circle */
        fprintf (file_ptr, "\". Circle %d\"\n", i); /* label center w/ circle */

        /* print circle circumference at intervals   = 360 / SUBDIVISIONS     */
        for (j=0; j < 360 + (360 / SUBDIVISIONS); j += 360 / SUBDIVISIONS)
        {
            if (TRACE) printf (" %f %f\n",
                     (circle_ptr->circle.center.x  +
                        circle_ptr->circle.radius * cos (j * PI / 180.0)),
                     (circle_ptr->circle.center.y  +
                        circle_ptr->circle.radius * sin (j * PI / 180.0)));
            fprintf (file_ptr, " %f %f\n",
                     (circle_ptr->circle.center.x  +
                        circle_ptr->circle.radius * cos (j * PI / 180.0)),
                     (circle_ptr->circle.center.y  +
                        circle_ptr->circle.radius * sin (j * PI / 180.0)));
            if (circle_ptr->circle.radius == 0.0)
               break;                  /* only one point needed in point case   */
        }
        if (TRACE) printf ("\" \"\n"); /* quoted blank to delimit this circle */
        fprintf (file_ptr, "\" \"\n"); /* quoted blank to delimit this circle */
    }
    fclose (file_ptr);
    if (TRACE) printf ("\n*** graph_world complete\n");
    return;
}
/*-----------------------------------------------------------------------------*/

void   output_path (path, filename)

       /* Output path in AUV data file format, appended to filename            */

       Path  *path;
       char  *filename;
{
    FILE        *file_ptr;
    int          i, j;              /* indices                                 */
    Path_list   *path_ptr;          /* index pointer to legs on the path       */
    if (TRACE) printf ("\n*** output_path start\n");

    path_ptr = path->path_list;       /* point to first leg of path            */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("output_path:  file open failure!", NONFATAL);
        return;
    }
    if (path->label != NULL)
    {
        fprintf (file_ptr, "\nPath      %s\n\n", path->label); /* path header */
        if (TRACE) printf ("\nPath      %s\n\n", path->label); /* path header */
    }
    else
    {
        fprintf (file_ptr, "\nPath      \n\n");
        if (TRACE) printf ("\nPath      \n\n");
    }

    /* print starting line segment data                                        */
```

```
fprintf (file_ptr, "Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                   path->initial_segment.point1.x,
                   path->initial_segment.point1.y,
                   DEFAULT_Z,
                   path->initial_segment.point2.x,
                   path->initial_segment.point2.y,
                   DEFAULT_Z);
if (TRACE) printf ("Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                   path->initial_segment.point1.x,
                   path->initial_segment.point1.y,
                   DEFAULT_Z,
                   path->initial_segment.point2.x,
                   path->initial_segment.point2.y,
                   DEFAULT_Z);
pooltime++;
fprintf (file_ptr, "              time %4.1f\n", pooltime);
if (TRACE) printf ("              time %4.1f\n", pooltime);

/* print all succeeding arc / line segment combinations                    */
for (i=1; i <= path->degree; ++i, path_ptr = path_ptr->next)
{
 if (path_ptr->arc.rotation != 0)   /* don't print arc if nothing's there */
   {
    fprintf (file_ptr, "Arc      %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f %2i ",
             path_ptr->arc.circle.center.x,
             path_ptr->arc.circle.center.y,
             DEFAULT_Z,
             path_ptr->arc.circle.radius,
             degrees (normalize2 (path_ptr->arc.angle1)),
             degrees (normalize2 (path_ptr->arc.angle2)),
             path_ptr->arc.rotation);

    pooltime++;
    if      (path_ptr->arc.rotation == CW)
             fprintf (file_ptr, "= CW      time %4.1f\n", pooltime);
    else if (path_ptr->arc.rotation == CCW)
             fprintf (file_ptr, "= CCW     time %4.1f\n", pooltime);
    else if (path_ptr->arc.rotation == CENTER)
             fprintf (file_ptr, "= CENTER  time %4.1f\n", pooltime);
    else     fprintf (file_ptr, "          time %4.1f\n", pooltime);

    if (TRACE) printf ("Arc      %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f %2i ",
             path_ptr->arc.circle.center.x,
             path_ptr->arc.circle.center.y,
             DEFAULT_Z,
             path_ptr->arc.circle.radius,
             degrees (normalize2 (path_ptr->arc.angle1)),
             degrees (normalize2 (path_ptr->arc.angle2)),
             path_ptr->arc.rotation);

    if (TRACE)
    {
        if      (path_ptr->arc.rotation == CW)
                 printf ("= CW      time %4.1f\n", pooltime);
        else if (path_ptr->arc.rotation == CCW)
                 printf ("= CCW     time %4.1f\n", pooltime);
        else if (path_ptr->arc.rotation == CENTER)
                 printf ("= CENTER  time %4.1f\n", pooltime);
        else     printf ("          time %4.1f\n", pooltime);
    )
   }

    fprintf (file_ptr, "Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
             path_ptr->segment.point1.x,
             path_ptr->segment.point1.y,
             DEFAULT_Z,
             path_ptr->segment.point2.x,
             path_ptr->segment.point2.y,
             DEFAULT_Z);

    if (TRACE) printf ("Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                       path_ptr->segment.point1.x,
                       path_ptr->segment.point1.y,
                       DEFAULT_Z,
                       path_ptr->segment.point2.x,
                       path_ptr->segment.point2.y,
                       DEFAULT_Z);
    pooltime++;
    fprintf (file_ptr, "              time %4.1f\n", pooltime);
    if (TRACE) printf ("              time %4.1f\n", pooltime);

}
```

```
    fprintf (file_ptr, "\n");
    if (TRACE) printf ("\n*** output_path complete\n");
    fclose  (file_ptr);
    return;
}
/*-------------------------------------------------------------------------------*/

void   output_world (circle_world, filename)

       /* Output circle world using AUV data file format, appended to filename*/

       Circle_world *circle_world;
       char         *filename;
{
    FILE        *file_ptr;
    int          i, j;              /* indices                                  */

    Circle_list *circle_ptr;        /* index pointer to current circle          */
    circle_ptr = circle_world->circle_list; /* point to first circle in world */
    if (TRACE) printf ("\n*** output_world start\n");

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("output_world:  file open failure!", NONFATAL);
        return;
    }
    fprintf (file_ptr,
                    "\n       Circle_World Shortest Path Determination\n\n\n");
    fprintf (file_ptr, "\nData specifications are according to ");
    fprintf (file_ptr, "the AUV Data Dictionary.\n\n\n");
    if (TRACE)
    {
       printf (         "\n       Circle_World Shortest Path Determination\n\n\n");
       printf (         "\nData specifications are according to ");
       printf (         "the AUV Data Dictionary.\n\n\n");
    }
    fprintf (file_ptr, "Point    %8.2f %8.2f %7.2f ", circle_world->start.x,
                                        circle_world->start.y,
                                        DEFAULT_Z);
    fprintf (file_ptr, "    Start\n");
    if (TRACE) printf ("Point    %8.2f %8.2f %7.2f ", circle_world->start.x,
                                        circle_world->start.y,
                                        DEFAULT_Z);
    if (TRACE) printf ("    Start\n");

    fprintf (file_ptr, "Point    %8.2f %8.2f %7.2f ", circle_world->goal.x,
                                        circle_world->goal.y,
                                        DEFAULT_Z);
    fprintf (file_ptr, "    Goal\n\n");
    if (TRACE) printf ("Point    %8.2f %8.2f %7.2f ", circle_world->goal.x,
                                        circle_world->goal.y,
                                        DEFAULT_Z);
    if (TRACE) printf ("    Goal\n\n");

    if (TRACE) printf ("\n*** output_world circle_world->degree = %d \n",
                       circle_world->degree);
    for (i=1; i <= circle_world->degree; ++i, circle_ptr = circle_ptr->next)
    {
        /* print circle center and radius                                       */
        fprintf (file_ptr, "Circle  %8.2f %8.2f %8.2f %8.2f \n",
                           circle_ptr->circle.center.x,
                           circle_ptr->circle.center.y,
                           DEFAULT_Z,
                           circle_ptr->circle.radius);
        if (TRACE) printf ("Circle  %8.2f %8.2f %8.2f %8.2f \n",
                           circle_ptr->circle.center.x,
                           circle_ptr->circle.center.y,
                           DEFAULT_Z,
                           circle_ptr->circle.radius);
    }
    if (TRACE) printf ("\n*** output_world complete\n");
    fprintf (file_ptr, "\n");
    fclose  (file_ptr);
    return;
}
/*-------------------------------------------------------------------------------*/

void center_graph_window (filename, xminptr, xmaxptr, yminptr, ymaxptr,
                          magnification)

/* Center (square off) the graph window so printed circles aren't distorted   */
```

```
    char   *filename;               /* graph filename for input & output     */
    double *xminptr, *xmaxptr,      /* output values (also appended to file) */
           *yminptr, *ymaxptr,
           magnification;           /* amount to magnify graph window bounds */
{
    FILE    *file_ptr;
    int      col;                   /* current column being used in line     */
    char     line [80];             /* input line string of characters       */
    double   x, y;                  /* input values from current line        */
    double   xmin, xmax, ymin, ymax, /* min/max values                       */
             deltax, deltay;        /* x, y max-min differences              */

    if (TRACE) printf ("\n*** center_graph_window start\n");

    xmin =   HUGE_VAL;              /* Typical graph ticks are 5 units apart */
    ymin =   HUGE_VAL;
    xmax = - HUGE_VAL;
    ymax = - HUGE_VAL;

    if ((file_ptr = fopen (filename, "r")) == ((FILE *) 0))
    {
        error ("center_graph_window:  file initial open failure!", NONFATAL);
        return;
    }
    if (TRACE) printf ("\n*** center_graph_window:  %s is open\n", filename);

    while ((fgets (line, 81, file_ptr) != NULL))    /* read next line of file */
    {
           col = 0;
           while (line [col] == ' ') col++;        /* skip initial blanks     */
           if (isdigit(line[col]) || (line [col] == '-') || (line [col] == '+')
                                  || (line [col] == '.'))
           {
              if (sscanf (line+col,"%lf",&x) != 1) break; /* get x gracefully*/

              while (isdigit ((int) line [col])   || (line [col] == '-') ||
                                 (line [col] == '.') || (line [col] == '+'))
                     col++;                          /* skip digits of x        */
              while ((line[col] == ' ') || (line [col] == ','))
                     col++;                          /* skip characters before y */

              if (sscanf (line+col,"%lf",&y) != 1) break;  /* get y gracefully*/
              if (xmin > x) xmin = x;
              if (ymin > y) ymin = y;
              if (xmax < x) xmax = x;
              if (ymax < y) ymax = y;
              if (TRACE)printf("\n*** center_graph_window loop check:");
              if (TRACE)printf(" (x, y)=(%6.2f, %6.2f)  ", x, y);
              if (TRACE)printf(" (xmin, ymin)=(%6.2f, %6.2f)", xmin, ymin);
              if (TRACE)printf(" (xmax, ymax)=(%6.2f, %6.2f)", xmax, ymax);
           } /* only lines beginning with numeric values are checked          */
    } /* end while */

    if (TRACE) printf ("\n*** center_graph_window while loop done\n        ");
    if (TRACE) printf ("(xmin, ymin) = (%6.2f, %6.2f)  ", xmin, ymin);
    if (TRACE) printf ("(xmax, ymax) = (%6.2f, %6.2f)  ", xmax, ymax);

    /* Now square off the extremes so no distortion occurs                    */
    if      ((ymax - ymin) < (xmax - xmin))
              ymax = ymin  + (xmax - xmin);
    else if ((xmax - xmin) < (ymax - ymin))
              xmax = xmin  + (ymax - ymin);

    if (magnification != 1.0) /* stretch out graph window boundaries          */
    {
       deltax = xmax - xmin;
       deltay = ymax - ymin;
       xmin -= deltax * (magnification - 1.0) / 2.0;
       xmax += deltax * (magnification - 1.0) / 2.0;
       ymin -= deltay * (magnification - 1.0) / 2.0;
       ymax += deltay * (magnification - 1.0) / 2.0;
    }

    if (TRACE) printf ("\n*** center_graph_window square-off ");
    if (TRACE) printf ("and magnification complete:\n        ");
    if (TRACE) printf ("(xmin, ymin) = (%6.2f, %6.2f)  ", xmin, ymin);
    if (TRACE) printf ("(xmax, ymax) = (%6.2f, %6.2f)  ", xmax, ymax);
    if (TRACE) printf ("\n        ");
    if (TRACE) printf ("magnification = %4.2f", magnification);

    *xminptr = xmin;        /* set returned values using pointer indirection  */
    *yminptr = ymin;
```

```
    *xmaxptr = xmax;
    *ymaxptr = ymax;

    fclose (file_ptr);
    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("center_graph_window:  file re-open failure!", NONFATAL);
        return;
    }
    /* append min/max points to file to square off graph boundaries           */
    if ((xmin !=   HUGE_VAL) && (ymin !=   HUGE_VAL))
        fprintf (file_ptr, " %8.2f %8.2f\n\" \"\n", xmin, ymin);
    if ((xmax != - HUGE_VAL) && (ymax != - HUGE_VAL))
        fprintf (file_ptr, " %8.2f %8.2f\n\" \"\n", xmax, ymax);

    fclose (file_ptr);
    if (TRACE) printf ("\n*** center_graph_window complete\n");

    return;
}
/*-------------------------------------------------------------------------*/

void retrieve_circle_world (circle_world)

    Circle_world  *circle_world;
{
    char           filename [40], line [120];
    FILE          *file_ptr;
    double         x, y, r;
    Point          start_point, goal_point, center_point;
    Circle         circle;

    if (TRACE) printf ("\n*** retrieve_circle_world begin\n");

    printf ("\n\nEnter the name of the circle_world file to retrieve:  ");
    scanf ("%s", filename);

    while ((file_ptr = fopen (filename, "r")) == ((FILE *) 0))
    {
        error ("retrieve_circle_world file open failure...\n  ", NONFATAL);
        printf ("\nPlease reenter the name of the circle_world file ");
        printf ("to be retrieved:  ");
        scanf ("%s", filename);
    }

    while (TRUE) /* loop to get start point */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read start point */
       {
           error ("retrieve_circle_world start point read failure", NONFATAL);
           return;
       }
       else  if (strcmp (line, "Point") == 0)
             {
                 fscanf (file_ptr, "%lf %lf", &x, &y);
                 start_point = make_point (x, y);
                 if (TRACE)
                     printf ("\n*** Start point = (%4.2f, %4.2f)\n", x, y);
                 break;
             }
    }
    while (TRUE) /* loop to get goal point */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read goal point */
       {
           error ("retrieve_circle_world goal point read failure", NONFATAL);
           return;
       }
       else  if (strcmp (line, "Point") == 0)
             {
                 fscanf (file_ptr, "%lf %lf", &x, &y);
                 goal_point = make_point (x, y);
                 if (TRACE)
                     printf ("\n*** Goal point  = (%4.2f, %4.2f)\n", x, y);
                 break;
             }
    }
    create_circle_world (start_point, goal_point, circle_world);

    while (TRUE) /* loop to get next circle */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read next circle */
```

```
        {
            break;
        }
        else  if (strcmp (line, "Circle") == 0)
              {
                  fscanf (file_ptr, "%lf %lf %*lf %lf", &x, &y, &r);
                  center_point = make_point (x, y);
                  circle       = make_circle (center_point, r);
                  if (TRACE)
                  printf ("\n*** Circle     = (%4.2f, %4.2f, %4.2f)\n", x, y, r);
                  add_circle_to_world (circle, circle_world);
              }
    }
    if (TRACE)
    {
        printf ("\n*** circle_world start point = (%4.2f, %4.2f)\n",
                 circle_world->start.x, circle_world->start.y);
        printf ("\n*** circle_world goal  point = (%4.2f, %4.2f)\n",
                 circle_world->goal.x, circle_world->goal.y);
        printf ("\n*** circle_world degree      = %d\n",
                 circle_world->degree);
    }
    if (TRACE) printf ("\n*** retrieve_circle_world complete\n");
    fclose (file_ptr);

    return;
}
/*-----------------------------------------------------------------------------*/
```

```
/*****************************************************************************
*                                                                            *
*   Filename:    c_search.c   circle_search                                   *
*                                                                            *
*   Purpose:     Determine single source shortest path using Dijkstra or A-star *
*                search algorithms for a circle_world robotics project.      *
*                                                                            *
*   Reference:   Advanced Robotics class notes, Dr. Yutaka Kanayama          *
*                                                                            *
*   Author:      Don Brutzman                                                *
*                                                                            *
*   Date:        10 February 92                                              *
*                                                                            *
*   Language:    ANSI C                                                      *
*                                                                            *
*   Compile:     cc -g -c  c_search -lm                                      *
*                                                                            *
*   Comments:    circle_world is a set of routines for mobile robot modeling *
*                   and two-dimensional path planning.                       *
*                                                                            *
*                All obstacles are modeled as circles.                       *
*                                                                            *
*   Status:      Shortest-path solution using Dijkstra's Algorithm or        *
*                  or A-star search complete.                                *
*                                                                            *
*****************************************************************************/

/* Include next 3 statements in this order for any circle_world application:  */

#ifndef  CIRCLE.C_INCLUDED
#include "circle.c"
#endif


#define MAX_CIRCLES      100    /* determines size search & tangent matrices  */

#define START            0      /* indices in search and tangent matrices     */
#define GOAL             1

#define MARKED           1      /* boolean values                             */
#define UNMARKED         0

#define DIJKSTRA         1      /* Search types                               */
#define A_STAR           2

#define TRACE            FALSE  /* Enable trace printf statements, c_search.c */



/****** List of circle_search Data Structures and Functions ******************/
/*

     Data Structures            Data Types and Member Labels     Matrix name
     ___________________        ____________________________     ___________

     Tangent_matrix_element     int      visible;                tangents [n][n]
                                double   segment_cost;
                                double   leaving_angle;
                                double   landing_angle;

     Search_matrix_element      int      mark;                   search  [n]
                                int      predecessor;
                                double   least_cost;
                                double   landing_angle;


   Functions                                Parameters
   _________                                __________

   initialize_tangents_and_search_matrices  (search_type, circle_world,
                                             all_tangents)

   update_unmarked_least_costs              (marked_node, circle_world)

   unmarked_vertices

   unmarked_circle_with_minimum_cost        (search_type, plot_each_leg,
                                             circle_world)
```

```
                                        [includes A-star evaluation function
                                         of arc & segment distance to goal]

   build_best_path                      (circle_world)

   circle_search                        (search_type, plot_each_leg,
                                         circle_world,
                                         best_path, all_tangents)

*/


/****** Circle_search Data Structures, Type Definitions & Global Variables ****/

/*---------------------------------------------------------------------------*/

typedef struct Tangent_matrix_element_type
{
  int          visible;         /* whether segment is VISIBLE, NONVISIBLE,    */
                                /*   or TANGENTIAL                            */
  double       segment_cost;    /* cost of this tangent segment               */
  double       leaving_angle;   /* angle leaving from i_th circle             */
  double       landing_angle;   /* angle landing on   j_th circle             */
}
  Tangent_matrix_element;

/*---------------------------------------------------------------------------*/

static Tangent_matrix_element  tangents [2*MAX_CIRCLES+1][2*MAX_CIRCLES+1];

  /* tangents */    /*  square global tangent_matrix of tangents and costs   */
                    /*  zeroth element corresponds to Start point S          */
                    /*  first  element corresponds to Goal  point G          */
                    /*  second element corresponds to circle 1 CW  side      */
                    /*  third  element corresponds to circle 1 CCW side      */
                    /*  (2n)   element corresponds to circle d CW  side      */
                    /*  (2n+1) element corresponds to circle d CCW side      */
                    /*         where d = circle_world->degree                */

                    /*  column i circles are the leaving circles             */
                    /*  row    j circles are the landing circles             */

                    /*  circle CW  sides have even indices                   */
                    /*  circle CCW sides have odd  indices                   */

                    /*  Note that travel is impossible between the           */
                    /*    CW & CCW sides of an individual circle.            */

                    /*  Also note that tangents [i][j] and tangents [j][i]   */
                    /*    are determined independently.  This allows the use */
                    /*    of a directionally dependent cost function         */
                    /*    without loss of generality.                        */

/*---------------------------------------------------------------------------*/



/*---------------------------------------------------------------------------*/

typedef struct Search_matrix_element_type
{
  int          mark;            /* boolean whether shortest path is found     */
  int          predecessor;     /* circle or (start point) preceding circle   */
  double       least_cost;      /* least cost found so far to this circle     */
  double       landing_angle;   /* angle landing on this circle               */
}
  Search_matrix_element;

/*---------------------------------------------------------------------------*/

static Search_matrix_element  search [2*MAX_CIRCLES+1];

  /* search */      /*  matrix used to perform single source shortest path   */
                    /*    search (Dijkstra's algorithm)                      */

/*---------------------------------------------------------------------------*/

static int n;       /* Number of active indices in tangents & search matrices */
                    /*    n = 2 * circle_world->degree + 1;                   */

static int total_visible;
```

```
                    /* total number of tangents visible in the set of all      */
                    /*    possible external & cross-tangents, between the      */
                    /*    start point, goal point and circles in circle_world  */

static int leg_number = 0;
                /*  occurrence number of latest leg with least cost         */
/*-------------------------------------------------------------------------*/


/****** Circle_search Function Declarations ********************************/

/*-------------------------------------------------------------------------*/

void  initialize_tangents_and_search_matrices (search_type, circle_world,
                                               all_tangents)

      int           search_type;          /* DIJKSTRA or A_STAR              */
      Circle_world *circle_world;
      Path         *all_tangents;

{
    int             i, j, model, mode2;   /* indices and rotation modes       */
    int             START_GOAL, visible2; /* logical checks                   */
    double          angle;                /* working variables                */
    Point           point1,    point2;
    Segment         segment;
    Circle          circle0,   circle1,   circle2;
    Arc             arc0;
    Configuration  *config1,  *config2;

    point1.x = circle_world->start.x - 0.3; /* offset x, y for tangents label */
    point1.y = circle_world->start.y - 0.8;
    segment  = make_segment (point1, point1);

    all_tangents->label      = " All visible tangents between circles included";
    all_tangents->degree          = 0;
    all_tangents->initial_segment = segment;
    all_tangents->path_list       = ((Path_list *) 0);  /* NULL */

    config1 = (Configuration *) malloc (sizeof(Configuration));
    config2 = (Configuration *) malloc (sizeof(Configuration));

    total_visible = 0;                    /* initialize visible tangent count */


    /*-------------------- Tangents matrix initialization --------------------*/

    if (TRACE)
    {
       printf ("\n\n");
       printf ("_____________________________________________");
       printf ("_____________________________________________");
       printf ("\n\n");
    }
    printf("\n\nCommencing least cost path determination using ");
    if      (search_type == DIJKSTRA) printf ("Dijkstra ");
    else if (search_type == A_STAR)   printf ("A-star ");
    else                              printf ("circle_");
    printf("search...\n\n\n");

    for (i = START; i <= n;  ++i)
    {
        if      (i == START)
                 circle1 = make_circle (circle_world->start, 0.0);
        else if (i == GOAL)
                 circle1 = make_circle (circle_world->goal,  0.0);
        else     circle1 = find_circle (i/2, circle_world);

        if      ((i == START) || (i == GOAL))
                 model = CENTER;          /* circle_world start & goal points  */
        else if (i == (i / 2) * 2)
                 model = CW;              /* even => CW side of circle         */
        else
                 model = CCW;             /* odd => CCW side of circle         */

        if (TRACE) printf ("\n____Tangent_______");
        if ((TRACE == TRUE) && (i >= 10)) printf ("_");
        if (TRACE) printf (
               "  Visibility __Circles__  ________Coordinates_________ _Cost_");

        for (j = START; j <= n;  ++j) /*-------------------------------------*/
        {
```

```
if      (j == START)
         circle2 = make_circle (circle_world->start, 0.0);
else if (j == GOAL)
         circle2 = make_circle (circle_world->goal,  0.0);
else     circle2 = find_circle (j/2, circle_world);

if      ((j == START) || (j == GOAL))
         mode2 = CENTER;      /* circle_world start & goal points  */
else if (j == (j / 2) * 2)
         mode2 = CW;          /* even => CW side of circle         */
else
         mode2 = CCW;         /* odd => CCW side of circle         */

/*-------------------------------------------------------------------*/

if   ((i == START && j == GOAL) || (i == GOAL && j == START))
{
      START_GOAL = TRUE;
      tangents [i][j].segment_cost = distance (circle_world->start,
                                               circle_world->goal);
}
else  START_GOAL = FALSE;

if ((i/2 == j/2) && (START_GOAL == FALSE))
/* IMPOSSIBLE diagonal case:  same circle, Start-Start, Goal-Goal */
{
  tangents [i][j].visible         = NONVISIBLE;
  if (((i == GOAL)&&(j == GOAL)) || ((i == START)&&(j == START)))
     tangents [i][j].segment_cost = 0.0;
  else
     tangents [i][j].segment_cost = HUGE_VAL;
  tangents [i][j].leaving_angle   = 0.0;
  tangents [i][j].landing_angle   = 0.0;
)
else                              /* all other non-IMPOSSIBLE cases    */
{
  circle_tangent (circle1, circle2, model, mode2, config1, config2);

  point1  = circumference_point (circle1, config1->tangent.angle);
  point2  = circumference_point (circle2, config2->tangent.angle);
  segment = make_segment (point1, point2);

  tangents [i][j].visible = visible (point1, point2, circle_world);

  /* Calculate costs if VISIBLE & save segment cost value          */
  if  ( (tangents [i][j].visible == VISIBLE) ||
        ((tangents [i][j].visible == TANGENTIAL) && TANGENTS_OK))
  {
       tangents [i][j].segment_cost = segment_cost (segment);
       total_visible++;

       /* add a segment to path containing only tangents           */
       angle       = orientation (point1, point2) + (PI/2.0);
       circle0     = make_circle (point1, 0.0);
       arc0        = make_arc    (circle0, angle, angle, 0);
       augment_path (arc0, segment, all_tangents);
  }

  else  /* segment is NONVISIBLE, no cost calculations required    */
  {
       tangents [i][j].segment_cost = HUGE_VAL;
  }
  if (j == GOAL) /* get distance-to-goal A* evaluation function    */
                 /*     regardless of visibility                  */
      tangents [i][j].segment_cost = segment_cost (segment);

  tangents [i][j].leaving_angle = config1->tangent.angle;
  tangents [i][j].landing_angle = config2->tangent.angle;
}

if (TRACE)
{
  printf("\n*** tangent [%d][%d] ", i, j);

  if      (j < 10) printf (" ");
  if      ((i/2 == j/2) && (START_GOAL == FALSE))
           printf ("IMPOSSIBLE");
  else if (tangents [i][j].visible == VISIBLE)
           printf ("   VISIBLE");
  else if (tangents [i][j].visible == NONVISIBLE)
           printf ("NONVISIBLE");
  else if (tangents [i][j].visible == TANGENTIAL)
```

```
                    printf ("TANGENTIAL");

                if      (i == 0) printf (" S ");       /* leaving circle #      */
                else if (i == 1) printf (" G ");
                else             printf (" %d ", i/2);

                if      (model == CW)     printf (" CW..");
                else if (model == CCW)    printf ("CCW..");
                else if (model == CENTER) printf (" PT..");
                else                      printf ("     ");

                if      (j == 0) printf ("S ");       /* landing circle #      */
                else if (j == 1) printf ("G ");
                else             printf ("%d ", j/2);

                if      (mode2 == CW)     printf ("CW ");
                else if (mode2 == CCW)    printf ("CCW");
                else if (mode2 == CENTER) printf ("PT ");
                else                      printf ("   ");
                if ((i/2 != j/2) || (START_GOAL == TRUE))
                {
                   printf (" (%5.2f,%5.2f)..(%5.2f,%5.2f)",
                            point1.x, point1.y, point2.x, point2.y);

                   if (( tangents [i][j].visible == VISIBLE) ||
                       ( START_GOAL == TRUE) ||
                       ((tangents [i][j].visible == TANGENTIAL) && TANGENTS_OK) ||
                       ((tangents [i][j].visible == NONVISIBLE) && (j == GOAL)))
                      printf (" %4.1f", tangents [i][j].segment_cost);
                }

                /* Change TANGENTIAL to VISIBLE or NONVISIBLE as appropriate    */
                if     ((tangents [i][j].visible == TANGENTIAL) && TANGENTS_OK)
                         tangents [i][j].visible =  VISIBLE;
                else if (tangents [i][j].visible == TANGENTIAL)
                         tangents [i][j].visible =  NONVISIBLE;
            }
        }                       /* for j loop complete */
    if (TRACE) printf ("\n");
    }                           /* for i loop complete */

    /*---------------------- Search matrix initialization ----------------------*/

    for (i = START; i <= n;  ++i)
    {
        search [i].mark          = UNMARKED;
        search [i].predecessor   = NULL;   /* impossible initialization value */
        search [i].least_cost    = HUGE_VAL;
        search [i].landing_angle = 0.0;
    }
    search [START].mark          = MARKED; /* Start point found by definition */
    search [START].predecessor   = START;  /* initialize remaining slots      */
    search [START].least_cost    = 0.0;
    search [START].landing_angle = 0.0;

    printf ("\n\nTangent and search matrix initializations are complete.\n");
    printf ("\n\n%d out of %d potential tangents (%3.1f %%) are usable.\n\n",
            total_visible, (n * n + 1),
            100.0 * (float) total_visible / (float) (n * n + 1));
    return;
}
/*-------------------------------------------------------------------------------*/

void update_unmarked_least_costs (marked_node, circle_world)

     int            marked_node;
     Circle_world  *circle_world;
{
    int             new,              /* index for checking each new node     */
                    rotation1,        /* rotations for first & second circles */
                    rotation2,
                    sign_second_arc;  /* MINUS PLUS or ZERO                   */
    double          new_cost,
                    angle1,           /* marked circle landing angle          */
                    angle2,           /* marked circle leaving angle          */
                    angle3,           /* new circle landing angle best so far */
                    angle4;           /* new circle landing angle latest      */
    Arc             first_arc,        second_arc;
    Circle          first_arc_circle, second_arc_circle;

    if (TRACE)
    {
```

```
    printf  ("\n*** update_unmarked_least_costs visible from node %d ",
             marked_node);
    if      (marked_node == START)
             printf ("(start point)");
    else if (marked_node == GOAL)
             printf ("(goal point)");
    else     printf ("(circle %d", (marked_node/2));
    if      ((marked_node != GOAL) && (marked_node != START) &&
             ((marked_node/2)*2 == marked_node))
             printf (" CW left side)");
    else if ((marked_node != GOAL) && (marked_node != START))
             printf (" CCW right side)");
    printf (":");
}

for (new = GOAL; new <= n; ++new)          /* loop through all elements    */
{
    /* Calculate total cost to new circle via marked_node circle          */

    if      (marked_node == START)
             first_arc_circle = make_circle (circle_world->start, 0.0);
    else if (marked_node == GOAL)
             first_arc_circle = make_circle (circle_world->goal,  0.0);
    else     first_arc_circle = find_circle (marked_node/2, circle_world);

    angle1 = search   [marked_node].landing_angle;
    angle2 = tangents [marked_node][new].leaving_angle;

    if (marked_node == (marked_node / 2) * 2)
        rotation1 = CW;                /* even node => circle CW  side    */
    else
        rotation1 = CCW;               /* odd  node => circle CCW side    */

    first_arc = make_arc (first_arc_circle, angle1, angle2, rotation1);

    /* Calculate cost difference due to different landing points          */

    if      (new == START)
             second_arc_circle = make_circle (circle_world->start, 0.0);
    else if (new == GOAL)
             second_arc_circle = make_circle (circle_world->goal,  0.0);
    else     second_arc_circle = find_circle (new/2, circle_world);

    angle3 = search   [new].landing_angle;            /* prior best angle */
    angle4 = tangents [marked_node][new].landing_angle;   /* latest angle */

    if (new == (new / 2) * 2)
        rotation2 = CW;                /* even node => circle CW  side    */
    else
        rotation2 = CCW;               /* odd  node => circle CCW side    */

    second_arc = make_arc (second_arc_circle, angle3, angle4, rotation2);

    if      ((search [new].predecessor == NULL) ||
             (search [new].predecessor == START))
             sign_second_arc = ZERO;   /* no arc exists if no predecessor */

    else if ((precede (angle4, angle3) && (rotation2 == CW )) ||
             (precede (angle3, angle4) && (rotation2 == CCW)))
             sign_second_arc = MINUS;/*this arc_cost applies to prior best*/
    else
             sign_second_arc = PLUS; /*this arc_cost applies to current   */

    /*--------------------------------------------------------------------*/

    /* After all this work we finally can add up costs for comparison     */

    new_cost = search [marked_node].least_cost +
               fabs (arc_cost (first_arc)) +
               tangents [marked_node][new].segment_cost;

    /*--------------------------------------------------------------------*/

    if (TRACE)
    {
        if      (new == GOAL) printf ("\n    Goal:   ",  new);
        else                  printf ("\n    node %d: ", new);
        if      (new < 10)    printf (" ");

        if      ((marked_node/2 == new/2) && (new != START)
                                          && (new != GOAL))
                 printf ("IMPOSSIBLE");
```

```
            else if (tangents [marked_node][new].visible == VISIBLE)
                     printf ("   VISIBLE");
            else if (tangents [marked_node][new].visible == NONVISIBLE)
                     printf ("NONVISIBLE");
            else if (tangents [marked_node][new].visible == TANGENTIAL)
                     printf ("TANGENTIAL");

            if      (search [new].mark == MARKED)   printf ("   MARKED");
            else if (search [new].mark == UNMARKED) printf (" UNMARKED");
            printf  (" prior best cost ");

            if ((search [new].least_cost >= 0.0) &&
                (search [new].least_cost < 10.0))        printf (" ");
            if ((search [new].least_cost == HUGE_VAL))   printf (" ");
            printf  ("%f", search [new].least_cost);
            if ((search [new].least_cost == HUGE_VAL))   printf ("inite");

            if ((tangents [marked_node][new].visible == VISIBLE)  &&
                (search [new].mark                   == UNMARKED))
            {
                printf (", new cost");
                if ((new_cost >= 0.0) &&
               (new_cost + (fabs (arc_cost (second_arc)) * sign_second_arc < 10.0)))
                                                        printf (" ");
                if (new_cost == HUGE_VAL)               printf (" ");
                printf ("%f",
                   new_cost + (fabs (arc_cost (second_arc)) * sign_second_arc));
                if (new_cost == HUGE_VAL) printf ("inite");
             }
        }

        /*-------------------------------------------------------------------------*/

        /*  Compare and replace if new_cost is better than current cost       */

        if ((tangents [marked_node][new].visible == VISIBLE)  &&
            (search [new].mark                   == UNMARKED) &&
            (new_cost >= 0.0) &&
            (search [new].least_cost >
                  new_cost + (fabs (arc_cost (second_arc)) * sign_second_arc)))
           {
              search [new].least_cost    = new_cost;
              search [new].predecessor   = marked_node;
              search [new].landing_angle =
                           tangents [marked_node][new].landing_angle;
           }
    }
    if (TRACE)
    {
       printf ("\n*** update_unmarked_least_costs via node %d ", marked_node);
       if      (marked_node == START)
                printf ("(start point)");
       else if (marked_node == GOAL)
                printf ("(goal point)");
       else     printf ("(circle %d", (marked_node/2));
       if      ((marked_node != GOAL) && (marked_node != START) &&
                ((marked_node/2)*2 == marked_node))
                printf (" CW left side)");
       else if ((marked_node != GOAL) && (marked_node != START))
                printf (" CCW right side)");
       printf (" complete.\n");
    }

    return;
}
/*-------------------------------------------------------------------------------*/

int unmarked_vertices ()     /* boolean value for remaining unmarked vertices */

{
    int i;

    for (i = GOAL;  i <= n; ++i)   /* Don't check START 0, default is MARKED  */
    {
        if (search [i].mark == UNMARKED)
        {
           if (TRACE)
           {
              printf ("\n*** unmarked_vertices check result = TRUE; ");
              printf ("unmarked vertices exist.\n");
           }
           return TRUE;
```

```
        }
    }
    if (TRACE)
    {
       printf ("\n*** unmarked_vertices check result = FALSE; ");
       printf ("no unmarked vertices remain.");
    }
    return FALSE;
}
/*-------------------------------------------------------------------------*/

int unmarked_circle_with_minimum_cost (search_type, plot_each_leg, circle_world)

    int             search_type,          /* DIJKSTRA or A_STAR              */
                    plot_each_leg;        /* TRUE or FALSE                   */
    Circle_world   *circle_world;         /* input:  all circles             */

    /* includes A-star evaluation function of arc & segment distance to goal  */

{
    int             i, least_circle,      /* working variables declaration   */
                    model, mode2, last, last2;
    double          min_cost, eval1, eval2;
    static char     label [10];
    Point           point1, point2;
    Circle          circle1, circle2;
    Arc             arc1, arc2;
    Segment         segment;
    Configuration *config1,*config2;
    Path           *current_leg;

    config1  = (Configuration *) malloc (sizeof (Configuration));
    config2  = (Configuration *) malloc (sizeof (Configuration));

    min_cost     = HUGE_VAL;
    least_circle = START;                 /* this initial value isn't possible*/

    if (TRACE)
        printf ("\n*** unmarked_circle_with_minimum_cost determination:");

    for (i = GOAL; i <= n; ++i)     /* Don't check START 0, default is MARKED  */
    {

        if (search_type == A_STAR) /* then calculate evaluation functions     */
        {
           /*-- Determine eval1: previous best min_cost circle "least_circle" */

           last =  search [least_circle].predecessor;
           if      (least_circle == START)
                    circle1 = make_circle (circle_world->start, 0.0);
           else if (least_circle == GOAL)
                    circle1 = make_circle (circle_world->goal,  0.0);
           else     circle1 = find_circle (least_circle/2, circle_world);

           if      ((least_circle == START) || (least_circle == GOAL))
                    model = CENTER;
           else if (least_circle == (least_circle/2) * 2)
                    model = CW;
           else     model = CCW;

           arc1  = make_arc (circle1,
                             tangents [last][least_circle].landing_angle,
                             tangents [least_circle][GOAL].leaving_angle, model);
           eval1 = min_cost + arc_cost (arc1) +
                   tangents [least_circle][GOAL].segment_cost;

           /*-- Determine eval2:  current min_cost circle "i" ----------------*/

           last2 =  search [i].predecessor;
           if      (i == START)
                    circle2 = make_circle (circle_world->start, 0.0);
           else if (i == GOAL)
                    circle2 = make_circle (circle_world->goal,  0.0);
           else     circle2 = find_circle (i/2, circle_world);

           if      ((i == START) || (i == GOAL))
                    mode2 = CENTER;
           else if (i == (i/2) * 2)
                    mode2 = CW;
           else     mode2 = CCW;

           arc2 = make_arc (circle2,
```

```
                        tangents [last2][i].landing_angle,
                        tangents [i][GOAL].leaving_angle, mode2);
            eval2 = search [i].least_cost + arc_cost (arc2) +
                    tangents [i][GOAL].segment_cost;
         }
         /* Determine if current circle beats previous best min_cost circle ---*/

         if (search [i].mark == UNMARKED) /* Evaluate which node has min cost  */
         {
            if (((search_type==DIJKSTRA) && (min_cost > search [i].least_cost))
            ||  ((search_type==A_STAR) && (min_cost!=HUGE_VAL) && (eval1>eval2))
            ||  ((search_type==A_STAR) && (min_cost==HUGE_VAL) &&
                                          (search [i].least_cost != HUGE_VAL)))
            {
                min_cost     = search [i].least_cost;
                least_circle = i;
            }
         }
         /*-- TRACE statements for each loop ---------------------------------*/
         if ((TRACE) && (search [i].mark == UNMARKED))
         {
            if      (i == GOAL) printf ("\n    Goal:   ",  i);
            else                printf ("\n    node %d: ", i);
            if      (i < 10)    printf (" ");

            if (search_type == DIJKSTRA)
            {
               printf ("current cost ");
               if ((search [i].least_cost >= 0.0) &&
                   (search [i].least_cost < 10.0))       printf (" ");
               if     (search [i].least_cost == HUGE_VAL) printf (" ");
               printf ("%f", search [i].least_cost);
               if     (search [i].least_cost == HUGE_VAL) printf ("inite");
            }
            else            /* A_STAR */
            {
               printf ("distance to goal ");
               if ((tangents [i][GOAL].segment_cost + arc_cost (arc2) >= 0.0) &&
                   (tangents [i][GOAL].segment_cost + arc_cost (arc2) < 10.0))
                   printf (" ");
               if (tangents [i][GOAL].segment_cost== HUGE_VAL)
                   printf (" ");
               printf ("%f", tangents [i][GOAL].segment_cost + arc_cost (arc2));
               if     (tangents [i][GOAL].segment_cost== HUGE_VAL)
                       printf ("inite");
               printf (", current evaluation function ");
               if ((eval2 >= 0.0) && (eval2 < 10.0)) printf (" ");
               if ((eval2 == HUGE_VAL))              printf (" ");
               printf ("%f", eval2);
               if     (eval2 == HUGE_VAL)            printf ("inite");
printf ("\narc_cost (arc2) = %f", arc_cost (arc2));
            }
         }
      }
      /*-------------------------------------------------------------------------*/

      if (plot_each_leg == TRUE)   /* plot this minimum cost leg in graph file  */
      {
          last = search [least_circle].predecessor;
          if      (last == START)
                   circle1 = make_circle (circle_world->start, 0.0);
          else if (last == GOAL)
                   circle1 = make_circle (circle_world->goal,  0.0);
          else     circle1 = find_circle (last/2, circle_world);

          if      ((last == START) || (last == GOAL))
                   model = CENTER;
          else if (last == (last/2) * 2)
                   model = CW;
          else     model = CCW;

          if      (least_circle == START)
                   circle2 = make_circle (circle_world->start, 0.0);
          else if (least_circle == GOAL)
                   circle2 = make_circle (circle_world->goal,  0.0);
          else     circle2 = find_circle (least_circle/2, circle_world);

          if      ((least_circle == START) || (least_circle == GOAL))
                   mode2 = CENTER;
          else if (least_circle == (least_circle/2) * 2)
                   mode2 = CW;
          else     mode2 = CCW;
```

```
        circle_tangent (circle1, circle2, mode1, mode2, config1, config2);
        circle1 = config1->tangent.circle;
        circle2 = config2->tangent.circle;
        point1  = circumference_point (circle1, config1->tangent.angle);
        point2  = circumference_point (circle2, config2->tangent.angle);
        segment     = make_segment (point1, point2);

        current_leg = create_path (segment);
        leg_number++;
        sprintf (label, "leg %d", leg_number);
        current_leg->label = label;
        graph_path (current_leg, circle_world, GRAPH_FILENAME);
    }

    /*- TRACE statements for completion -------------------------------------*/
    if (TRACE)
    {
        printf ("\n*** unmarked_circle_with_minimum_cost:  node %d",
                    least_circle);
        printf (" with current cost %f;", min_cost);
        printf ("\n    ");
        if   (least_circle == GOAL) printf ("(GOAL)");
        else                        printf ("      ");
        printf (" node %d is now marked and has node %d",
                    least_circle, search [least_circle].predecessor);
        printf (" as its predecessor.\n");
    }
    return least_circle;
}
/*-------------------------------------------------------------------------*/

void *build_best_path (circle_world, best_path)

      Circle_world  *circle_world;    /* work from goal to start & build path */
      Path          *best_path;       /*   using predecessor & least_cost     */
                                      /*   results from circle_search         */
{
    int         circle,    predecessor,
                current,   rotation;
    double      angle1,    angle2;
    static char label2 [40];

    Point       point1,    point2;
    Segment     segment;
    Circle      circle1,   circle2;
    Path_list  *path_ptr, *new_leg;

    if (TRACE)
    {
       printf ("\n\n");
       printf ("_____________________________________________");
       printf ("_____________________________________________");
       printf ("\n\n");
       printf("\n*** build_best_path:  work backwards from goal to start.\n");
    }

    best_path->path_list = NULL;
    best_path->degree    = 0;
    sprintf (label2, "       Best path (cost %4.1f)",
            search [GOAL].least_cost);
    best_path->label = label2;

    predecessor = GOAL;               /* initialize: algorithm begins at GOAL */
    point2      = circle_world->goal;

    while (predecessor != START)      /* proceed backward until START reached */
    {
        current     = predecessor;
        predecessor = search [predecessor].predecessor; /* take one step back */
        if (TRACE)
        {
            printf ("\n*** build best path:  predecessor = %d, current = %d",
                     predecessor, current);
            printf (", adding path leg to list.\n");
        }
        if (predecessor == START)     /* returned to START, best path built   */
        {
            point1  = circle_world->start;
            segment = make_segment (point1, point2);
            best_path->initial_segment = segment;
        }
        else                          /* construct & insert another path leg  */
```

```
        {
            if      (predecessor == START)
                    circle1 = make_circle (circle_world->start, 0.0);
            else if (predecessor == GOAL)
                    circle1 = make_circle (circle_world->goal,  0.0);
            else    circle1 = find_circle (predecessor/2, circle_world);

            if      (current == START)
                    circle2 = make_circle (circle_world->start, 0.0);
            else if (current == GOAL)
                    circle2 = make_circle (circle_world->goal,  0.0);
            else    circle2 = find_circle (current/2, circle_world);

            angle1  = search  [predecessor].landing_angle;
            angle2  = tangents[predecessor][current].leaving_angle;

            /* Determine rotation associated with starting circle             */
            if      ((predecessor == START) || (predecessor == GOAL) ||
                     (circle1.radius == 0.0))
                     rotation = 0;
            else if (predecessor == (predecessor / 2) * 2)
                     rotation = LEFT;
            else     rotation = RIGHT;

            new_leg = (Path_list *) malloc (sizeof (Path_list));
            new_leg->arc = make_arc (circle1, angle1, angle2, rotation);

            point1 = circumference_point
                        (circle1, tangents[predecessor][current].leaving_angle);
            point2 = circumference_point
                        (circle2, search [current].landing_angle);
            new_leg->segment = make_segment (point1, point2);

            /* set up for eventual construction of initial path segment       */
            point2 = circumference_point (circle1, angle1);

            /* insert new_leg at head of the existing path list               */
            path_ptr              = best_path->path_list;
            best_path->path_list  = new_leg;
            best_path->degree++;
            if (path_ptr != NULL)
                path_ptr->previous = new_leg;
            new_leg->previous     = NULL;
            new_leg->next         = path_ptr;
        }
    }

    if (TRACE)
    {
       printf ("\n*** build_best_path:  complete\n\n");
       printf ("_____________________________________________");
       printf ("_____________________________________________");
       printf ("\n\n");
    }

    printf ("\nThe best path from start to goal has cost = %f\n\n",
             search [GOAL].least_cost);
    printf ("\nThe best path includes arcs around %d circle obstacle",
             best_path->degree);
    if   (best_path->degree == 1) printf (".\n");
    else                          printf ("s.\n");
    printf("\n\nLeast cost path determination using circle_search complete.");
    printf ("\n\n");
    printf ("_____________________________________________");
    printf ("_____________________________________________");
    printf ("\n");

    return (best_path);
}
/*-------------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------------*/
/*                                                                               */
/* Determination of best path through circle_world using Dijkstra's or A-star */
/*   search algorithms for single-source shortest paths.                         */
/*                                                                               */
/* Reference:  Manber, Udi, _Introduction to Algorithms - A Creative          */
/*             Approach_, Addison-Wesley Publishing Company, Reading,         */
/*             Massachusetts, 1989.                                           */
/*                                                                               */
```

```
/*-------------------------------------------------------------------------*/

void circle_search (search_type, plot_each_leg, circle_world,
                    best_path, all_tangents)

     int             search_type,       /* DIJKSTRA is a greedy algorithm       */
                                        /* A_STAR  evaluates distance to goal   */
                     plot_each_leg;     /* TRUE or FALSE                        */
     Circle_world   *circle_world;      /* input:  all circles                  */
     Path           *best_path,         /* output: best path start to goal      */
                    *all_tangents;      /* output: all visible tangent segments */

{
    int             w;                 /* declaration for minimum cost node    */

    n = 2 * circle_world->degree + 1; /* initialize global total: 2 * circles */
                                       /*   (CW & CCW) plus START and GOAL     */
                                       /*   See declarations for index details */


/*- begin Dijkstra's or A-Star Search Algorithm ------------------------------*/


    initialize_tangents_and_search_matrices (search_type, circle_world,
                                             all_tangents);

    update_unmarked_least_costs (START, circle_world);

    while (unmarked_vertices ())

    {
        w = unmarked_circle_with_minimum_cost (search_type, plot_each_leg,
                                               circle_world);

        search [w].mark = MARKED;

        if (w == GOAL) break;

        update_unmarked_least_costs (w, circle_world);
    }

    build_best_path (circle_world, best_path);

    return;

}
/*-------------------------------------------------------------------------*/
```

```
/*****************************************************************************
*                                                                            *
*   Filename:   circtest.c   circle_world                                    *
*                                                                            *
*   Purpose:    Test program to evaluate circle world (circle.c & c_search.c)*
*               functionality for circle_world robotics project.             *
*                                                                            *
*   Reference:  Advanced Robotics class notes, Dr. Yutaka Kanayama           *
*                                                                            *
*   Author:     Don Brutzman                                                 *
*                                                                            *
*   Date:       10 February 92                                               *
*                                                                            *
*   Language:   ANSI C                                                       *
*                                                                            *
*   Compile:    cc -g -o circle_world circtest.c -lm                         *
*                                                                            *
*   Execution:  circle_world                                                 *
*                                                                            *
*   Graphing:   graph -b -g 1 -l "circle world" < circle.graph | lpr -g      *
*                                                                            *
*               Additional graph details are available using manual pages,   *
*                 i.e. 'man graph' and 'man plot'.                           *
*                                                                            *
*   Comments:   Circle_world is a set of routines for mobile robot modeling  *
*                 and two-dimensional path planning.                         *
*               Circle_search performs minimum cost path circle search using *
*                 Dijkstra's algorithm.                                      *
*               Circle_test allows entering circle_world data, computing     *
*                 external and cross-tangents, checking visiblility, and     *
*                 determining a least-cost path from start to goal.          *
*                                                                            *
*               All obstacles are modeled as circles.                        *
*                                                                            *
*   Status:     Near-optimal complexity, shortest path solution complete.    *
*                                                                            *
*               Segments and arcs sequentially numbered using time parameter *
*                 to optionally allow graphic visualization of search process*
*                                                                            *
*****************************************************************************/

/* Include the next 3 lines in this order for any circle_world application:   */

#ifndef  CIRCLE.C_INCLUDED
#include "circle.c"
#endif


#include "c_search.c"

#include <time.h>

/*---------------------------------------------------------------------------*/

main ()             /* circle_test */

/*---------------------------------------------------------------------------*/

/*  Declarations and initializations:                                        */

{
    double          x, y, r;
    double          xmin, xmax, ymin, ymax;  /* min & max values             */
    char            answer = 'y',
                    title    [80], /* string array for graph title           */
                    command [160], /* string array for system commands       */
                    date     [32], /* string for today's date                */
                    label    [40]; /* string for path label                  */
    time_t          today;

    int             model, mode2,  /* rotation directions                    */
                    i, j, k,       /* indices                                */
                    copies,
                    search_type,   /* DIJKSTRA or A_STAR search              */
                    plot_each_leg; /* TRUE or FALSE                          */

    Point           start,     goal;
    Point           point1,    point2,    point3,   point4;
    Segment         segment0,  segment1,  segment2;
    Circle          circle1,   circle2;
    Tangent         tangent1,  tangent2;
    Arc             arc1,      arc2;
```

```
    Configuration  *config1,  *config2;
    Path           *path0,    *path1,    *path2;
    Circle_list    *circle_ptr;
    Circle_world   *circle_world;

    /* instantiate default paths using a pseudo-segment; instantiate config's */
    point1.x     = 0.0;
    point1.y     = 0.0;
    segment1     = make_segment (point1, point1);
    path0        = create_path  (segment1);
    path1        = create_path  (segment1);
    path2        = create_path  (segment1);

    config1      = (Configuration *) malloc (sizeof (Configuration));
    config2      = (Configuration *) malloc (sizeof (Configuration));

    circle_world = (Circle_world  *) malloc (sizeof (Circle_world));

/*-----------------------------------------------------------------------------*/

/*  Introduction and file reset:                                               */

    printf ("\n\n");
    printf ("_____________________________________________");
    printf ("_____________________________________________");
    printf ("\n\n");
    printf ("\n          ");
    printf ("circle.c is a library of routines for mobile robot modeling ");
    printf ("\n          ");
    printf ("             and two-dimensional path planning.\n");
    printf ("\n          ");
    printf ("circle_search performs minimum cost path circle search using");
    printf ("\n          ");
    printf ("             Dijkstra or A-star search algorithms ");
    printf ("\n          ");
    printf ("             and Euclidean distance cost function.\n");
    printf ("\n          ");
    printf (" circle_world allows entering circle_world data, computing    ");
    printf ("\n          ");
    printf ("   external and cross-tangents, checking visibility, and   ");
    printf ("\n          ");
    printf ("      determining a least-cost path from start to goal.        ");
    printf ("\n\n");
    printf ("_____________________________________________");
    printf ("_____________________________________________");

    printf ("\n\n\nDo you want to retrieve a saved circle_world file?    ");
    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y')
    {
        retrieve_circle_world (circle_world);
        if  (circle_world != NULL)
        {
             point1.x = circle_world->start.x;
             point1.y = circle_world->start.y;
             point2.x = circle_world->goal.x;
             point2.y = circle_world->goal.y;
             printf ("\n\n   Start point = (%5.2f, %5.2f)", point1.x, point1.y);
             printf ("\n\n   Goal  point = (%5.2f, %5.2f)", point2.x, point2.y);
             printf ("\n\n   Circles in circle_world:  %d",
                     circle_world->degree);
       }
       else
       {
             printf ("\nRetrieval of the circle_world file was unsuccessful.");
             answer = 'n';
       }
    }

    printf ("\n\n");
    printf ("\nRemoving previous copies of circle_world output files:");

    printf  ("\nrm %s\n",      GRAPH_FILENAME);
    sprintf (command, "rm %s", GRAPH_FILENAME);
    system  (command);

    printf  ("rm %s\n",        AUV_FILENAME);
    sprintf (command, "rm %s", AUV_FILENAME);
    system  (command);

/*-----------------------------------------------------------------------------*/
```

```
/*  Enter circle_world start point, goal point and circle data:                */

    if (((answer != 'y') && (answer != 'Y')) || (circle_world == NULL))
    {
        printf ("\n\nEnter the start point x and y coordinates.\n\n");
        printf ("  start x = ");
        scanf  ("%lf", &x);              /* note %lf to convert to double precision  */
        printf ("  start y = ");
        scanf  ("%lf", &y);
        point1 = make_point (x, y);

        printf ("\nEnter the goal  point x and y coordinates.\n\n");
        printf ("  goal  x = ");
        scanf  ("%lf", &x);
        printf ("  goal  y = ");
        scanf  ("%lf", &y);
        printf ("\n");
        point2 = make_point (x, y);

        create_circle_world (point1, point2, circle_world);
        answer = 'y';
    }
    /* create a segment and a path direct from start to goal                    */

    start    = point1;
    goal     = point2;
    segment0 = make_segment (point1, point2);
    printf ("\n\nStraight-line path from start to goal cost = %4.2f\n",
            segment_cost (segment0));
    path0 = create_path (segment0);
    sprintf (label, "Straight line start to goal (cost = %4.2f)",
             segment_cost (segment0));
    path0->label = label;

    if ((answer == 'y' || answer == 'Y') && (circle_world->degree >= 1))
       /* true if circle_world file was read successfully                       */
    {
        scanf  ("%c", &answer); /* hack to clear carriage return from buffer  */
        printf ("\n\nDo you want to enter another circle?  ");
        scanf  ("%c", &answer);
    }

    while ((answer == 'y' || answer == 'Y') || (circle_world->degree < 1))
    {
        printf ("\n\nEnter circle # %d center coordinates & radius.\n\n",
                 circle_world->degree + 1);
        printf ("  circle # %d x = ", circle_world->degree + 1);
        scanf  ("%lf", &x);
        printf ("  circle # %d y = ", circle_world->degree + 1);
        scanf  ("%lf", &y);
        printf ("  circle # %d r = ", circle_world->degree + 1);
        scanf  ("%lf", &r);
        while (r < 0.0)
        {
            printf ("\nPlease enter a non-negative radius value:  ");
            scanf  ("%lf", &r);
        }
        printf ("\n");
        point1  = make_point (x, y);
        circle1 = make_circle (point1, r);
        add_circle_to_world (circle1, circle_world);

        scanf  ("%c", &answer); /* hack to clear carriage return from buffer  */
        printf ("Do you want to enter another circle?  ");
        scanf  ("%c", &answer);
    }
    printf ("\n");
    printf ("_____________________________________________");
    printf ("_____________________________________________");
    printf ("\n\n");

/*  Circle world data entry complete.                                          */

/*  Output circle world in .graph point pair format and .auv data format       */
    graph_world  (circle_world, GRAPH_FILENAME);
    output_world (circle_world, AUV_FILENAME);

/*-------------------------------------------------------------------------------*/

/*  Test visibility from start to goal, followed by point pairs of interest:  */

    if      (visible (circle_world->start, circle_world->goal, circle_world)
```

```
            == TRUE)
          printf ("\nThe start and goal points are VISIBLE to each other.\n");

    else if (visible (circle_world->start, circle_world->goal, circle_world)
            == TANGENTIAL)
         {
          printf ("\nThe start and goal points are VISIBLE to each other and ");
          printf ("\ntheir line segment is TANGENTIAL to one or more circles ");
          printf ("in circle_world.\n");
         }
    else  printf ("\nThe start & goal points are NONVISIBLE to each other.\n");
    printf ("\n\n");

    scanf  ("%c", &answer); /* hack to clear carriage return from buffer      */
    printf ("Do you want to check visibility between other pairs of points?  ");
    scanf  ("%c", &answer);

    while (answer == 'y' || answer == 'Y')
    {
        printf ("\n\nEnter the first  point coordinates.\n\n");
        printf ("   first  x = ");
        scanf  ("%lf", &x);
        printf ("   first  y = ");
        scanf  ("%lf", &y);
        point1  = make_point (x, y);
        printf ("\n\nEnter the second point coordinates.\n\n");
        printf ("   second x = ");
        scanf  ("%lf", &x);
        printf ("   second y = ");
        scanf  ("%lf", &y);
        point2  = make_point (x, y);

        printf("\n\nThe distance between the two points = %f\n\n",
               distance (point1, point2));

        printf("\nThe orientation between the two points = %4.2f degrees\n\n",
               degrees (orientation (point1, point2)));

        if      ((visible (point1, point2, circle_world) == VISIBLE) &&
                 (visible (point2, point1, circle_world) == VISIBLE))
          printf("\nThe first and second points are VISIBLE to each other.");

        else if ((visible (point1, point2, circle_world) == TANGENTIAL) &&
                 (visible (point2, point1, circle_world) == TANGENTIAL))
        {
         printf("\nThe first and second points are VISIBLE to each other and ");
         printf ("\ntheir line segment is TANGENTIAL to one or more circles ");
         printf ("in circle_world.\n");
        }
        else
          printf("\nThe first and second points are NONVISIBLE to each other.");

        printf ("\n\n\n");
        scanf  ("%c", &answer); /* hack to clear carriage return from buffer  */
      printf("Do you want to check visibility between a new pair of points?   ");
        scanf  ("%c", &answer);
    }
/*-------------------------------------------------------------------------------*/

/*  Calculate external, cross & center tangents between circles of interest:  */

    if (circle_world->degree >= 1)
    {
       printf ("\n\n");
       scanf  ("%c", &answer); /* hack to clear carriage return from buffer   */
    printf ("Do you want to check tangents between circles or points?        ");
       scanf  ("%c", &answer);
    )
    else
       answer = 'n';

    while (answer == 'y' || answer == 'Y')
    {
        printf ("\n\nEnter the number of the first  circle to check.\n");
        printf ("(use zero for start point, -1 for goal point)\n\n");
        printf ("  first circle is # ");
        scanf  ("%d", &i);
        if      (i ==  0) circle1 = make_circle (start, 0.0); /* start point */
        else if (i == -1) circle1 = make_circle (goal,  0.0); /* goal  point */
        else              circle1 = find_circle (i, circle_world);

        printf("\n\nEnter the traversal mode of the first  circle.\n ");
```

```
model = 2;
while ((model < -1) || (model > 1))
{
    printf("  (Valid mode values are CCW +1, CW -1, Center 0): ");
    scanf("%i", &model);
    printf ("\n");
}

printf ("\nEnter the number of the second circle to check.\n");
printf ("(use zero for start point, -1 for goal point)\n\n");
printf ("  second circle is # ");
scanf  ("%d", &j);
while (j == i)
{
    printf("\nPlease enter a circle number different from the first: ");
    scanf  ("%d", &j);
    printf ("\n");
}
if      (j ==  0) circle2 = make_circle (start, 0.0); /* start point */
else if (j == -1) circle2 = make_circle (goal,  0.0); /* goal  point */
else              circle2 = find_circle (j, circle_world);

printf("\nEnter the traversal mode of the second circle.\n");
mode2 = 2;
while ((mode2 < -1) || (mode2 > 1))
{
    printf("  (Valid mode values are CCW +1, CW -1, Center 0): ");
    scanf("%i", &mode2);
    printf ("\n");
}

circle_tangent (circle1, circle2, model, mode2, config1, config2);

/* Update circle1 and circle2 due to CENTER case possibility          */
circle1 = config1->tangent.circle;
circle2 = config2->tangent.circle;

point1 = circumference_point (circle1, config1->tangent.angle);
point2 = circumference_point (circle2, config2->tangent.angle);

printf("\n\n");
printf("         Configuration1                    Configuration2     ");
printf("    Modes\n");
printf("  ____________________________     ____________________________");
printf("__ _____\n\n");
printf("   x1      y1    angle1  orient1     x2     y2    angle2  ");
printf("orient2\n\n");
printf("%5.1f  %5.1f  %7.2f  %7.2f  %5.1f  %5.1f   %7.2f  %7.2f  ",
     point1.x, point1.y, degrees (config1->tangent.angle),
     degrees (config1->orientation),
     point2.x, point2.y, degrees (config2->tangent.angle),
     degrees (config2->orientation));
if      (model == LEFT)   printf("L..");
else if (model == RIGHT)  printf("R..");
else if (model == CENTER) printf("C..");
else                      printf("?..");
if      (mode2 == LEFT)   printf("L\n\n");
else if (mode2 == RIGHT)  printf("R\n\n");
else if (mode2 == CENTER) printf("C\n\n");
else                      printf("?\n\n");

printf("\n\nThe distance between the two tangent points = %f\n\n",
       distance (point1, point2));

if       ((visible (point1, point2, circle_world) == TRUE) &&
          (visible (point2, point1, circle_world) == TRUE))
  printf ("\nThe first and second points are VISIBLE to each other.");

else if  ((visible (point1, point2, circle_world) == TANGENTIAL) &&
          (visible (point2, point1, circle_world) == TANGENTIAL))
  printf("\nThe first and second points are VISIBLE to each other and ",
         "\ntheir line segment is TANGENTIAL to one or more circles in",
         " circle_world.");
else
  printf("\nThe first and second points are NONVISIBLE to each other.");

/* Create the tangent 'path' starting with a pseudo-segment           */
segment1 = make_segment (circle1.center, circle1.center);
path1    = create_path (segment1);
arc1     = make_arc (circle1, config1->tangent.angle,
                             config1->tangent.angle,  model);
segment2 = make_segment (point1, point2);
```

```
        augment_path (arc1, segment2, path1);

        /* Subsequent graph_path & output_path calls append data to the files. */
        graph_path  (path1, circle_world, GRAPH_FILENAME);
        output_path (path1, AUV_FILENAME);
        printf ("\n\n\nThis tangent has been added to the output files.");

        printf ("\n\n\n");
        scanf  ("%c", &answer); /* hack to clear carriage return from buffer   */
        printf ("Do you want to test another set of circle tangents?  ");
        scanf  ("%c", &answer);
        printf ("\n");
    }

/*-------------------------------------------------------------------------------*/

/*  Determine search type and whether to plot each least cost leg found:      */

    answer = '*';
    while ((answer != 'd') && (answer != 'D') &&
           (answer != 'a') && (answer != 'A'))
    {
       if (answer != '*')
           printf ("\n*** Please answer D for Dijkstra or A for A-star ...\n");
       printf ("\n\n");
       scanf  ("%c", &answer); /* hack to clear carriage return from buffer*/
    printf ("Do you want a Dijkstra search or an A-star search?              ");
       scanf  ("%c", &answer);
       printf ("\n\n");
       if (answer == 'd' || answer == 'D') search_type = DIJKSTRA;
       if (answer == 'a' || answer == 'A') search_type = A_STAR;
    }

    scanf  ("%c", &answer); /* hack to clear carriage return from buffer      */
    printf ("Do you want to plot and number each least cost leg found?      ");
    scanf  ("%c", &answer);
    printf ("\n");

    if   (answer == 'y' || answer == 'Y')
         plot_each_leg = TRUE;
    else  plot_each_leg = FALSE;

/*-------------------------------------------------------------------------------*/

    circle_search (search_type, plot_each_leg, circle_world, path1, path2);

/*-------------------------------------------------------------------------------*/

    /* path1 is now least cost path, path2 is all tangents                    */

    scanf  ("%c", &answer); /* hack to clear carriage return from buffer      */
   printf ("\n\nDo you want to include the start to goal line on the graph?  ");
    scanf  ("%c", &answer);

    if (answer == 'y' || answer == 'Y')
    {
       graph_path  (path0, circle_world, GRAPH_FILENAME);
       output_path (path0, AUV_FILENAME);
    }

    scanf  ("%c", &answer);
   printf ("\n\nDo you want to include the least cost path in the output?    ");
    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y')
    {
        graph_path  (path1, circle_world, GRAPH_FILENAME);
        output_path (path1, AUV_FILENAME);
    }

    scanf  ("%c", &answer);
   printf ("\n\nDo you want to include ALL circle tangents in the output?    ");
    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y')
    {
        graph_path  (path2, circle_world, GRAPH_FILENAME);
        output_path (path2, AUV_FILENAME);
    }

    /* square off the graph data    */
    center_graph_window (GRAPH_FILENAME, &xmin, &xmax, &ymin, &ymax,
                         GRAPH_STRETCH);
```

```
/*-------------------------------------------------------------------------*/

    /* Check whether circle world is modeled in the NPS pool:               */
    scanf  ("%c", &answer);
   printf ("\n\nDo you want the NPS pool superimposed on the graph?         ");
    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y') /* invert x-axis, draw pool lines */
    {
        xmin     = 145.0;
        xmax     = -15.0;
        ymin     = -50.0;
        ymax     = 110.0;

        point1   = make_point   (  0.0,  0.0);
        point2   = make_point   (127.5,  0.0);
        point3   = make_point   (127.5, 67.5);
        point4   = make_point   (  0.0, 67.5);

        segment0 = make_segment (point1, point2);
        path0    = create_path  (segment0);
        graph_path (path0, circle_world, GRAPH_FILENAME);

        segment0 = make_segment (point2, point3);
        path0    = create_path  (segment0);
        graph_path (path0, circle_world, GRAPH_FILENAME);

        segment0 = make_segment (point3, point4);
        path0    = create_path  (segment0);
        graph_path (path0, circle_world, GRAPH_FILENAME);

        segment0 = make_segment (point4, point1);
        path0    = create_path  (segment0);
        graph_path (path0, circle_world, GRAPH_FILENAME);
    }

/*-------------------------------------------------------------------------*/

/*  Plot circle_world, tangents and path map using sunview sunplot function:  */

    scanf  ("%c", &answer);
   printf ("\n\nDo you want a sunview sunplot of the circle_world plotted?   ");
    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y')
    {
        scanf  ("%c", &answer);
        sprintf (command,
                 "graph -b -g 1  -x %f %f -y %f %f  <  %s  | sunplot -s -c 800",
                 xmin, xmax, ymin, ymax, GRAPH_FILENAME);
        printf ("\n\n%s\n\n\n", command);
        system  (command);
    }
    else scanf  ("%c", &answer);

/*-------------------------------------------------------------------------*/

/*  Print circle_world, tangents and path map using Unix graph function:      */

   printf ("\n\nDo you want a hard copy of the circle_world graph plotted?   ");

    scanf  ("%c", &answer);
    if (answer == 'y' || answer == 'Y')
    {
   printf ("\n\nHow many copies do you want printed?                        ");
        scanf  ("%d", &copies);
        if ((copies < 0) || (copies > 20))
        {
            printf ("\nThe allowable range of copies is (0..20).");
   printf ("\n\nHow many copies do you want printed?                        ");
            scanf  ("%d", &copies);
        }

        today = time (NULL);
        strftime (date, 32, "%d %B %Y", localtime (&today));
        scanf   ("%c", &answer);
        printf  ("\n\nEnter a title to be printed on the graph:  \n\n");
        gets    (title);
        sprintf (command,
"graph -b -g 1  -l \"%s    %s\"   -x %f %f -y %f %f < %s | lpr -g -h -#%d -Pap2"
                ,title, date, xmin, xmax, ymin, ymax, GRAPH_FILENAME, copies);
        printf ("\n\n%s\n\n\n", command);
        system  (command);
    }
```

APPENDIX F. OBTAINING NPS AUV INTEGRATED SIMULATOR PROGRAMS SOURCE CODE

NPS AUV Integrated Simulator graphics simulation program, sonar classification expert system and circle world path planning source code can be obtained on Internet via anonymous FTP. Figure F.1 shows an example of how to obtain these files.

```
% ftp taurus.cs.nps.navy.mil
Connected to taurus.cs.nps.navy.mil.
220 taurus FTP server (SunOS 4.1) ready.
Name (taurus.cs.nps.navy.mil:brutzman): anonymous
331 Guest login ok, send ident as password.
Password: your_name_here
230 Guest login ok, access restrictions apply.
ftp> cd pub
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> get auvsim.tar.Z
local: auvsim.tar.Z remote: auvsim.tar.Z
200 PORT command successful.
150 Binary data connection for auvsim.tar.Z (131.120.1.20,1250) (814519 bytes).
226 Binary Transfer complete.
814519 bytes received in 1.38 seconds (576.32 Kbytes/s)
ftp> quit
221 Goodbye.

% uncompress auvsim.tar.Z

% tar xf auvsim.tar

% ls
AUV                 auvsonar.clp        m35.d               nps_rightpool.off
AUVReadme           auvsonar.log        m35_gyro.auv        point.off
Makefile            c_search.c          m35_raw.auv         pool.auv
NPS_AUV             circle.auv          materials.off       pool_cylinder.off
NPS_AUV.c           circle.c            mine.off            pool_lightsl.off
NPS_AUV_fn.c        circle20.auv        mine2.off           pool_lights2.off
auv_lights.off      circtest.c          mine3.off           poolsnapshot
auv_mtls.off        cylinder.off        minehunt.auv        screensnapshot
auvsim              floor.off           nps_auv.off         showsgibwonly.c
auvsim.c            hull.off            nps_farpool.off     sphere.off
auvsim.h            loop.auv            nps_leftpool.off    test.auv
auvsim.tar          loop.d              nps_nearpool.off
auvsonar            m35.auv             nps_pool.off

% auvsim  /* to execute on your iris workstation! */
```

Figure F.1 Obtaining NPS AUV Integrated Simulator files via Internet

APPENDIX G. VIDEOTAPE DEMONSTRATION OF RESULTS

A videotape appendix is included to demonstrate operation and usefulness of the NPS AUV Integrated Simulator. The first video segment is the original segment submitted to the IEEE Robotics and Automation Conference 1992 (Brutzman Floyd Whalen 92). Video abstract and mission profile are included in Figures G.1 and G.2. Additional videotape demonstrations show graphics simulation program functionality and visualization of sonar classification, circle world path planning and minefield search applications.

Naval Postgraduate School Autonomous Underwater Vehicle

Charles A. Floyd, Donald P. Brutzman, and Russell Whalen
Computer Science Department, Naval Postgraduate School
Monterey California 93943 USA, brutzman@taurus.cs.nps.navy.mil

Abstract

The Naval Postgraduate School (NPS) Autonomous Underwater Vehicle (AUV) is an eight foot long, 387-pound untethered robot submarine designed for research in adaptive control, mission planning, navigation, mission execution, and post-mission data analysis. The NPS AUV has four active fixed-beam high-resolution ultrasonic sonars which point orthogonally ahead, downward and to port and starboard. Neutral buoyancy, eight plane surfaces and twin propellers allow precise maneuverability.

Simulation programs running on Iris three-dimensional graphics workstations are used to evaluate NPS AUV software and predict system performance prior to each mission. During simulation a complete hydrodynamics model accurately represents physical response characteristics through six degrees of freedom.

The videotaped NPS AUV test mission was performed in the Olympic-size NPS swimming pool and programmed to include waypoint maneuvering, sonar ranging, and a full pool traversal while recording pertinent sensor and posture values at a 10 Hz data rate.

Graphics simulations can replay in real time actual data collected in the pool. The taped playback demonstrates reconstruction and visualization of vehicle track, control systems dynamic response, logic and state changes, plotted locations of individual sonar returns, and expert system classification of detected objects.

Ongoing NPS AUV research is investigating linear and nonlinear control techniques, advanced sonar classification, failure mode analysis using neural networks, dynamic path and search planning, use of cross-body thrusters for hovering control, and alternate AUV operating architectures.

Figure G.1 NPS AUV video abstract

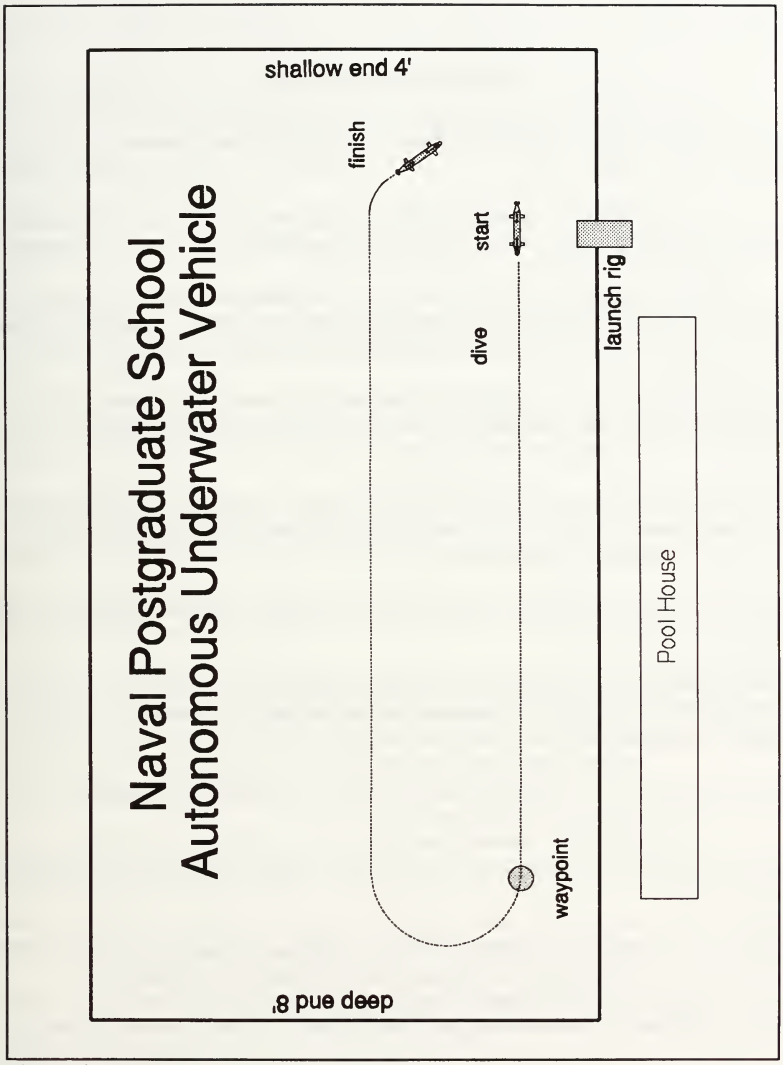


Figure G.2 Mission profile of NPS AUV video

LIST OF REFERENCES

Akman, Varol, *Unobstructed Shortest Paths in Polyhedral Environments*, Springer-Verlag, Berlin, 1987.

Arthur, VADM Stanley R. and Pokrant, Marvin, "Desert Storm at Sea," U.S. Naval Institute *Proceedings*, vol. 117 no. 5, May 1991, pp. 82-87.

Asano, Takano, Asano, Tetsuo, Guibas, Leonidas, Hershberger, John, and Imai, Hiroshi, "Visibility-Polygon Search and Euclidean Shortest Paths," *Proceedings of the 26th Symposium on Foundations of Computer Science*, 1985, pp. 155-164.

Badler, Norman I., Barsky, Brian A. and Zeltzner, David, ed., *Making Them Move: Mechanics, Control and Animation of Articulated Figures*, Morgan Kaufmann Publishers Inc., San Mateo, California, 1991.

Badr, Salah M., Byrnes, Ronald B., Brutzman, Donald P. and Nelson, Michael L., *Real-Time Systems*, technical report NPS-CS-92-004, Naval Postgraduate School, Monterey, California, February 1992.

Baerson, Kevin M., "Flight Lab Conquers Real-Time Unix," *Federal Computer Week*, December 2, 1991, p. 24.

Barrow, Theodore H., Yurchak, John M. and Zyda, Michael J., *Distributed Computer Communications In Support Of Real-Time Visual Simulations*, M.S. Thesis, Naval Postgraduate School, Monterey, California, September 1988.

Besl, P.J. and Jain, R.C., "Three-Dimensional Object Recognition," *Computing Surveys*, vol. 17 no. 1, March 1985, pp. 77-145.

Blackman, Maurice, *The Design of Real-Time Applications*, John Wiley and Sons Ltd., London, 1976.

Blidberg, D.R., Chappell, S., Jalbert, J., Turner, R., Sedor, G. and Eaton, P., "The EAVE AUV Program at the Marine Systems Engineering Laboratory," *Proceedings of 1st IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 1990, pp. 33-42.

Bobrow, Daniel G., "Dimensions of Interaction," *AI Magazine*, vol. 12 no. 3, Fall 1991, pp. 64-80.

Brooke, Tom, "The Art of Production Systems," *AI Expert*, vol. 7 no. 1, January 1992, pp. 30-35.

Brooks, Frederick P. Jr., "Grasping Reality through Illusion: Interactive Graphics Serving Science," included in "Implementing and Interacting with Real-time Microworlds," course 29, *ACM SIGGRAPH Conference*, Boston, Massachusetts, 31 July-4 August 1989, pp. 3-1 through 3-11.

Brutzman, Donald P. and Compton, Mark A., "AUV Research at the Naval Postgraduate School," *Sea Technology*, vol. 32 no. 12, December 1991, pp. 35-40.

Brutzman, Donald P., Floyd, Charles A. and Whalen, Russell, "Naval Postgraduate School Autonomous Underwater Vehicle," *Video Proceedings of the IEEE International Conference on Robotics and Automation 92*, Nice, France, May 1992.

Brutzman, Donald P., Kanayama, Yutaka and Zyda, Michael J., "Integrated Simulation for Rapid Development of Autonomous Underwater Vehicles", *Proceedings of the IEEE Oceanic Engineering Society Conference AUV 92*, Washington DC, June 1992.

Brutzman, Donald P., Compton, Mark A. and Kanayama, Yutaka, "Autonomous Sonar Classification using Expert Systems," draft article, *OCEANS 92* conference, Oceanic Engineering Society of the IEEE, Newport, Rhode Island, October 26-29, 1992.

Burke, JOC(SW) Kip, "More Than an Eye in the Sky," *Surface Warfare*, November/December 1991, pp. 8-9.

Byrnes, R.B., MacPherson, D.L., Kwak, S.H., Nelson, M.L. and McGhee, R.B., "An Experimental Comparison of Hierarchical and Subsumption Software Architectures for Control of an Autonomous Underwater Vehicle," presented at IEEE Oceanic Engineering Society Symposium on Autonomous Underwater Vehicles, Washington DC, June 2-3, 1992.

Canny, John F., *The Complexity of Robot Motion Planning*, The MIT Press, Cambridge, Massachusetts, 1988.

Comer, Douglas E., *Internetworking with TCP/IP Volume I: Principles, Protocols and Architecture*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1991.

Compton, LCDR Mark A., *Minefield Search and Object Recognition for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

Compton, LCDR Mark A., "Modeling the Sonar Environment," unpublished paper, Naval Postgraduate School, Monterey, California, September 1991.

Corkill, Daniel, "Blackboard Systems," *AI Expert*, vol. 6 no. 9, September 1991, pp. 40-47.

Cramer, Bill, "Writing Real-Time Programs under UNIX," *Dr. Dobb's Journal*, vol. 13 no. 6, June 1988, pp. 18-29.

Dasgupta, Partha, LeBlanc, Richard J., Jr., Ahamad, Mustaque, and Ramachandran, Umakishore, "The Clouds Distributed Operating System," *Computer*, November 1991, pp. 34-44.

Davis, Daniel, "Control of MBARI ROV Camera and Tools over a Network," *Proceedings of 1st IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 137-142.

Deitel, Harvey M., *An Introduction to Operating Systems*, Addison-Wesley Publishing Co. Ltd., Reading, Massachusetts, 1990.

Dibble, Peter, *OS-9 Insights: An Advanced Programmers Guide to OS-9/68000*, Microware Systems Corporation, Des Moines, Iowa, 1988.

Durfee, Edmund H. and Lesser, Victor R., "Planning to Meet Deadlines in a Blackboard-based Problem Solver," COINS Technical Report 87-07, *IEEE Tutorial on Real-Time Systems*, Computer Society Press of the IEEE, Washington DC, 1988.

Ethernet Installation Guide, Digital Equipment Corporation, Maynard, Massachusetts, 1983.

Etter, Paul C., *Underwater Acoustic Modeling: Principles, Techniques and Applications*, Elsevier Applied Science, London, England, 1991.

Falk, Howard, "Developers Target UNIX and Ada with Real-Time Kernels," *Computer Design*, vol. 27 no. 7, 1 April 1988, pp. 55-70.

Floyd, Charles A., *Design and Implementation of a Collision Avoidance System for the NPS Autonomous Underwater Vehicle (AUV II) Utilizing Ultrasonic Sensors*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

Floyd, Charles A., Kanayama, Yutaka, and Magrino, Christopher, "Underwater Obstacle Recognition using a Low-Resolution Sonar," *Proceedings of the Seventh International Symposium on Unmanned Untethered Submersible Technology*, University of New Hampshire, Durham, New Hampshire, September 1991, pp. 309-327.

GESPAC Inc., *Introduction to OS-9/68000*, class notes, Mesa, Arizona, 1989.

Giarratano, Joseph C., *CLIPS User's Guide*, NASA, Lyndon B. Johnson Space Center, January 1991.

Good, LT Michael R., *Design and Construction of a Second Generation AUV*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1989.

Gorey, Kevin, "Periodic Table of the Irises," Silicon Graphics Inc., Mountain View, California, February 1991.

Hamming, Richard W., *Numerical Methods for Scientists and Engineers*, second edition, McGraw-Hill Book Co., New York, 1973.

Hart, Peter E., Nilsson, Nils J. and Raphael, Bertram, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SSC-4 no. 2, July 1968, pp. 100-107.

Healey, A.J., McGhee, R.B., Cristi, R., Papoulias, F.A., Kwak, S.H., Kanayama, Y., and Lee, Y., "Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle", *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, California, October 23-26, 1990, pp. 177-186.

Healey, A.J., Papoulias, F.A., and MacDonald, G., "Design and Experimental Verification of a Model Based Compensator for Rapid AUV Depth Control," *Proceedings from the 6th International Symposium on Unmanned Untethered Submersible Technology*, University of New Hampshire, Durham, New Hampshire, June 12-14, 1989, pp. 458-474.

Hebert, Martial, Kanade, Takeo and Kweon, InSo, "3-D Vision Techniques for Autonomous Vehicles," *NSF Range Image Understanding Workshop*, 1988, pp. 273-337.

Hildebrand, Dan, "Message-Passing Operating Systems," *Dr. Dobb's Journal*, vol. 13 no. 6, June 1988, pp. 34-48.

Interview with Patrick Hale, DARPA UUV Project Manager, C.S. Draper Laboratories, Cambridge, Massachusetts by the author, December 11, 1991.

Interview between Anthony J. Healey, Chair, Mechanical Engineering Department, Naval Postgraduate School, and LCDR Mark Compton and the author, 9 August 91.

Interview between Robert B. McGhee, Chair, Computer Science Department, Naval Postgraduate School, and LCDR Mark Compton and the author, 8 August 91.

Iyengar, S. Sitharama and Elfes, Alberto, ed., *Autonomous Underwater Robots: Perception, Mapping and Navigation*, volume 1, IEEE Computer Society Press, Los Alamitos, California, 1991.

Jackson, Peter, *Introduction to Expert Systems*, Addison-Wesley Publishing Co. Inc., Workingham, England, 1991.

Jurewicz, CDR Thomas A., *A Real-Time Autonomous Underwater Vehicle Dynamic Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1990.

Kahaner, David K., "TRON (The Real-Time Operating System Nucleus)," *Scientific Information Bulletin*, vol. 16 no. 3, Office of Naval Research Asian Office, July-September 1991, pp. 11-19.

Kanayama, Yutaka, Noguchi, Tetsuo, and Hartman, Bruce, "Sonar Data Interpretation for Autonomous Mobile Robots," unpublished paper, Naval Postgraduate School, Monterey, California, 1990.

Kanayama, Yutaka and Noguchi, Tetsuo, "Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors," University of California Santa Barbara Department of Computer Science *Technical Report TRCS89-06*, February 1989.

Kanayama, Yutaka and Brutzman, Donald P., "Shortest Path Planning in a Circle World", unpublished paper, Naval Postgraduate School, Monterey, California, September 1991.

Kanayama, Y. and De Haan, G., "A Mathematical Theory of Safe Path Planning," Technical Report of Computer Science Department TRCS88-16, University of California at Santa Barbara, California, 1988.

Kanayama, Yutaka, "Advanced Robotics and Spatial Reasoning", class notes, Naval Postgraduate School, Monterey, California, May 1991.

Kasahara, Hironori, "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System," *IEEE Tutorial on Real-Time Systems*, Computer Society Press of the IEEE, Washington DC, 1988.

Kenny, Kevin B. and Lin, Kwei-Jay, "Building Flexible Real-Time Systems using the Flex Language," *Computer*, May 1991, pp. 70-78.

King, LT David Maurice and Prevatt, LCDR Richard Montgomery III, "Rapid Production of Graphical User Interfaces," Master's Thesis, Naval Postgraduate School, Monterey, California, December 1990.

Laumond, Jean-Paul, "Obstacle Growing in a Nonpolygonal World," *Information Processing Letters*, vol. 25 no. 1, April 1987, pp. 41-50.

Leatherman, Brent, "An Approach to Integration of Real-Time Software for Autonomous Underwater Vehicles," Masters Thesis, Naval Postgraduate School, Monterey California, September 1991.

Locke, John, *Physical Layout of the Computer Science Department Network*, general newsgroup posting, Naval Postgraduate School, Monterey, California, January 1991.

Lozano-Pérez, Tomás and Wesley, Michael A., "An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles," *Communications of the ACM*, vol. 22 no. 10, October 1979, pp. 560-570.

Luo, R.C. and Kay, M.G., "Multisensor Integration and Fusion in Intelligent Systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 19 no. 5, September/October 1989, pp. 901-931.

Makris, Dionysios, "Real-Time Scheduling and Synchronization for the Naval Postgraduate School Autonomous Underwater Vehicle," Masters Thesis, Naval Postgraduate School, Monterey California, December 1991.

Manber, Udi, *Introduction to Algorithms: A Creative Approach*, New York, Addison-Wesley, 1989, pp. 204-208.

Mathworks, Inc., *PC-MATLAB for MS-DOS Personal Computers*, South Natick, Massachusetts, 1989.

Mellichamp, Duncan A., ed., *Real-Time Computing with Applications to Data Acquisition and Control*, Van Nostrand Reinhold Co., New York, 1983.

Moravec, Hans, "The Stanford Cart and the CMU Rover," *Proceedings of the IEEE*, vol. 71 no. 7, July 1983, pp. 872-884.

Moravec, Hans P., *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*, Ph.D. Thesis, Report STAN-CS-80-813, Stanford University, September 1980.

Mullender, Sjoerd, *sunplot* computer program, Free University, Amsterdam, Netherlands, 1987.

Myers, Laura, "Silicon Graphics to introduce new workstation today," *The Herald*, Monterey, California, p. 4B, July 22, 1991.

NASA Software Technology Branch, *CLIPS Reference Manual*, Lyndon B. Johnson Space Center, Houston, Texas, 1991.

Ong, Seow Meng, *A Mission Planning Expert System with Three-Dimensional Path Optimization for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.

Pappas, George, Shotts, William, O'Brien, Mack and Wyman, William, "The DARPA/Navy Unmanned Undersea Vehicle Program," *Unmanned Systems*, vol. 9 no. 2, Spring 1991, pp. 24-30.

Payton, David W. and Bihari, Thomas E., "Intelligent Real-Time Control of Robotic Vehicles," *Communications of the ACM*, vol. 34 no. 8, August 1991, pp. 49-63.

Polmar, Norman, *The Ships and Aircraft of the U.S. Fleet*, Naval Institute Press, Annapolis, Maryland, 1987, p. 233.

Polmar, Norman, "Robot Submarines," U.S. Naval Institute *Proceedings*, vol. 117 no. 9, September 1991, pp. 122-123.

Preparata, Franco P., and Shamos, Michael Ian, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985, pp. 10-11.

Sacerdoti, Earl D., "Managing Expert System Development," *AI Expert*, vol. 6 no. 5, May 1991, pp. 26-33.

Schwartz, J.T. and Sharir, M., "A Survey of Motion Planning and Related Geometric Algorithms," *Artificial Intelligence*, vol. 37 no.s 1-3, December 1988, pp. 157-169.

Stallings, William, *Data and Computer Communications*, Macmillan Publishing Company, New York, 1988.

Stankovic, John A., ed., "Real-Time Computing Systems: The Next Generation," *IEEE Tutorial on Real-Time Systems*, Computer Society Press of the IEEE, Washington DC, 1988.

Stewart, W. Kenneth, "Three-Dimensional Modeling of Seafloor Backscatter from Sidescan Sonar for Autonomous Classification and Navigation," *Proceedings of the 6th International Symposium on Unmanned Untethered Submersible Technology*, University of New Hampshire, Durham, New Hampshire, June 1989, pp. 372-392.

Thalmann, Nadia Magnenat, and Thalmann, Daniel, *Computer Animation: Theory and Practice*, second edition, Springer-Verlag, Tokyo, Japan, 1990.

Welzl, Emo, "Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time", *Information Processing Letters*, vol. 20 no. 4, 10 May 1985, pp. 167-171.

West, RADM Ralph W., Jr., Superintendent, Naval Postgraduate School, memorandum to LCDR Mark Compton and the author, 16 August 91.

Wright, M. Lattimer, Green, Milton W., Fiegi, Gudrun, and Cross, Perry F., "An Expert System for Real-Time Control," *IEEE Tutorial on Real-Time Systems*, Computer Society Press of the IEEE, Washington DC, 1988.

Zyda, Michael J., "Object File Format", *Graphics and Video Laboratory*, unpublished course text, Naval Postgraduate School, Monterey, California, 2 April 1991, pp. 7.1-7.81.

Zyda, Michael J., McGhee, Robert B., Kwak, Sehung, Nordman, Douglas B., Rogers, Ray C. and Marco, David, "Three-Dimensional Visualization of Mission Planning and Control for the NPS Autonomous Underwater Vehicle," *IEEE Journal of Oceanic Engineering*, vol. 15 no. 3, July 1990, pp. 217-221.

Zyda, Michael J., Jurewicz, Thomas A., Floyd, Charles A. and McGhee, Robert B., "Physically Based Modeling of Rigid Body Motion in a Real-Time Graphical Simulator," unpublished paper, Naval Postgraduate School, Monterey, California, September 1991.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria Virginia 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey California 93943-5002	2
3.	Dr. Robert B. McGhee Code CS/Mz Chairman, Computer Science Department Naval Postgraduate School Monterey California 93943-5000	1
4.	Dr. Anthony J. Healey Code ME/Hy Chairman, Mechanical Engineering Department Naval Postgraduate School Monterey California 93943-5000	1
5.	Dr. Yutaka Kanayama Code CS/Ka Computer Science Department Naval Postgraduate School Monterey California 93943-5000	1
6.	Dr. Michael J. Zyda Code CS/Zk Computer Science Department Naval Postgraduate School Monterey California 93943-5000	1

7. CAPT Alan R. Beam USN — 1
DARPA UWO - PRC Inc.
1555 Wilson Boulevard
Suite 600
Arlington Virginia 22209

8. MAJ David Neyland USAF — 1
DARPO ASTO
3701 North Fairfax Drive
Arlington Virginia 22203

9. RADM George R. Sterner USN — 1
Program Executive Officer
Submarine Combat and Weapons Systems
Department of the Navy
Washington DC 20362-5101

10. Commander — 1
Naval Sea Systems Command
ATTN: CAPT William Shotts, PMO-403
Washington DC 20362-5101

11. Dr. Richard Guertin — 1
OP-09BC
Pentagon 4D386
Washington DC 20301-5000

12. Chief of Naval Research — 1
800 North Quincy Street
Arlington Virginia 22217-5000

13. Commander — 1
Submarine Development Squadron TWELVE
Naval Submarine Base
Groton Connecticut 06340

14. Commanding Officer — 1
Naval Underwater Systems Center
Newport Rhode Island 02841-5047

15. Commanding Officer 1
Naval Coastal Systems Center
Panama City Florida 32407-5000

16. Commander 1
Naval Surface Weapons Center
Dahlgren Virginia 22448-5000

17. Commanding Officer 1
David Taylor Research Center
Bethesda Maryland 20084-5000

18. Commander 1
Naval Oceans Systems Center
San Diego California 92152-5000

19. Mr. Randy Brill 1
Naval Oceans Systems Center
PO Box 997
Kailua Hawaii 96734-0996

20. Director 1
Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory
Washington DC 20375-5000

21. Mr. Patrick Hale 1
DARPA UUV Program Manager
C.S. Draper Laboratories
555 Technology Square
Cambridge Massachusetts 02139

22. Dr. D. Richard Blidberg 1
Marine Systems Engineering Laboratory
Marine Program Building
University of New Hampshire
Durham New Hampshire 03824-3525

23. Dr. James G. Bellingham 1
Sea Grant College Program
Massachusetts Institute of Technology
292 Main Street
Cambridge Massachusetts 02139

24. Dr. Dana R. Yoerger 1
Deep Submergence Laboratory
Department of Applied Ocean Physics and Engineering
Woods Hole Oceanographic Institute
Woods Hole Massachusetts 02543

25. Dr. Stanley Dunn 1
Advanced Marine Systems Group
Ocean Engineering Department
Florida Atlantic University
Boca Raton Florida 33431

26. CDR Charles A. Floyd 1
Computer Science Department
Chauvenet Hall 9F
U.S. Naval Academy
572 Holloway Road
Annapolis Maryland 21402-5002

27. Dr. Peter Purdue 1
Code OR/Pd
Chairman, Operations Research Department
Naval Postgraduate School
Monterey California 93943-5000

28. Dr. Se-Hung Kwak 1
Code CS/Kw
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

29. Dr. Luqi 1
Code CS/Lq
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

30. Dr. Yuh-Jeng Lee 1
Code CS/Le
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

31. Mr. David Pratt 1
Code CS/Pr
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

32. MAJ Ronald B. Byrnes USA 1
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

33. LCDR David L. MacPherson USN 1
Computer Science Department
Naval Postgraduate School
Monterey California 93943-5000

34. LCDR Donald P. Brutzman USN 1
Operations Research Department
Naval Postgraduate School
Monterey California 93943-5000

ACCOMPANY VHS 5000043 LOCATED AT
RESERVE DESK